Limbajul C#

PhD. Lucian Sasu

Cuprins

1 Platforma Microsoft .NET **11**
- 1.1 Prezentare generală . . . 11
- 1.2 Arhitectura platformei Microsoft .NET . . . 13
- 1.3 Componente ale lui .NET Framework . . . 14
  - 1.3.1 Common Intermediate Language . . . 14
  - 1.3.2 Common Language Specification . . . 14
  - 1.3.3 Common Language Runtime . . . 15
  - 1.3.4 Common Type System . . . 16
  - 1.3.5 Metadate . . . 17
  - 1.3.6 Assemblies . . . 18
  - 1.3.7 Assembly cache . . . 18
  - 1.3.8 Garbage collection . . . 19
- 1.4 Trăsături ale platformei .NET . . . 19

2 Tipuri predefinite, tablouri, string-uri **23**
- 2.1 Vedere generală asupra limbajului C# . . . 23
- 2.2 Tipuri de date . . . 25
  - 2.2.1 Tipuri predefinite . . . 25
  - 2.2.2 Tipuri valoare . . . 27
  - 2.2.3 Tipul enumerare . . . 32
- 2.3 Tablouri . . . 38
  - 2.3.1 Tablouri unidimensionale . . . 38
  - 2.3.2 Tablouri multidimensionale . . . 39
- 2.4 Şiruri de caractere . . . 43
  - 2.4.1 Expresii regulate . . . 45

3 Clase, instrucţiuni, spaţii de nume **47**
- 3.1 Clase – vedere generală . . . 47
- 3.2 Transmiterea de parametri . . . 51
- 3.3 Conversii . . . 57
  - 3.3.1 Conversii implicite . . . 57

3.3.2 Conversiile implicite ale expresiilor constante 59
3.3.3 Conversii explicite 59
3.3.4 Boxing şi unboxing 62
3.4 Declaraţii de variabile şi constante 64
3.5 Instrucţiuni C# 64
3.5.1 Declaraţii de etichete 64
3.5.2 Instrucţiuni de selecţie 65
3.5.3 Instrucţiuni de ciclare 66
3.5.4 Instrucţiuni de salt 68
3.5.5 Instrucţiunile try, throw, catch, finally 68
3.5.6 Instrucţiunile checked şi unchecked 69
3.5.7 Instrucţiunea lock 69
3.5.8 Instrucţiunea using 69
3.6 Spaţii de nume 70
3.6.1 Declaraţii de spaţii de nume 71
3.6.2 Directiva using 72
3.7 Declararea unei clase 76
3.8 Membrii unei clase 77
3.9 Constructori de instanţă 78
3.10 Câmpuri 78
3.10.1 Câmpuri instanţe 79
3.10.2 Câmpuri statice 79
3.10.3 Câmpuri readonly 80
3.10.4 Câmpuri volatile 80
3.10.5 Iniţializarea câmpurilor 81
3.11 Constante 81
3.12 Metode 82
3.12.1 Metode statice şi nestatice 83
3.12.2 Metode externe 83

4 Clase (continuare) 85
4.1 Proprietăţi 85
4.2 Indexatori 91
4.3 Operatori 97
4.3.1 Operatori unari 97
4.3.2 Operatori binari 99
4.3.3 Operatori de conversie 99
4.3.4 Exemplu: clasa Fraction 101
4.4 Constructor static 103
4.5 Clase imbricate 104
4.6 Destructori 107

4.7 Clase statice . . . 109
4.8 Specializarea şi generalizarea . . . 111
4.8.1 Specificarea moştenirii . . . 111
4.8.2 Apelul constructorilor din clasa de bază . . . 112
4.8.3 Operatorii *is* şi *as* . . . 113
4.9 Clase *sealed* . . . 114

5 Clase, structuri, interfeţe, delegaţi 115
5.1 Polimorfismul . . . 115
5.1.1 Polimorfismul parametric . . . 115
5.1.2 Polimorfismul ad–hoc . . . 115
5.1.3 Polimorfismul de moştenire . . . 116
5.1.4 *Virtual* şi *override* . . . 117
5.1.5 Modificatorul *new* pentru metode . . . 118
5.1.6 Metode *sealed* . . . 121
5.1.7 Exemplu folosind *virtual, new, override, sealed* . . . 122
5.2 Clase şi metode abstracte . . . 124
5.3 Tipuri parţiale . . . 125
5.4 Structuri . . . 127
5.4.1 Structuri sau clase? . . . 131
5.5 Interfeţe . . . 131
5.5.1 Clase abstracte sau interfeţe? . . . 137
5.6 Tipul delegat . . . 138
5.6.1 Utilizarea delegaţilor pentru a specifica metode la runtime 139
5.6.2 Delegaţi statici . . . 142
5.6.3 Multicasting . . . 143

6 Metode anonime. Evenimente. Excepţii. 147
6.1 Metode anonime . . . 147
6.2 Evenimente . . . 149
6.2.1 Publicarea şi subscrierea . . . 150
6.2.2 Evenimente şi delegaţi . . . 150
6.2.3 Comentarii . . . 156
6.3 Tratarea excepţiilor . . . 157
6.3.1 Tipul *Exception* . . . 157
6.3.2 Aruncarea şi prinderea excepţiilor . . . 158
6.3.3 Reîncercarea codului . . . 169
6.3.4 Compararea tehnicilor de manipulare a erorilor . . . 171
6.3.5 Sugestie pentru lucrul cu excepţiile . . . 172

7 Colecţii. Clase generice. 173
7.1 Colecţii . . . 173
7.1.1 Iteratori pentru colecţii . . . 176
7.1.2 Colecţii de tip listă . . . 177
7.1.3 Colecţii de tip dicţionar . . . 178
7.2 Crearea unei colecţii . . . 179
7.2.1 Colecţie iterabilă (stil vechi) . . . 179
7.2.2 Colecţie iterabilă (stil nou) . . . 182
7.3 Clase generice . . . 187
7.3.1 Metode generice . . . 188
7.3.2 Tipuri generice . . . 189
7.3.3 Constrângeri asupra parametrilor de genericitate . . . 190
7.3.4 Interfeţe şi delegaţi generici . . . 191
7.4 Colecţii generice . . . 192
7.4.1 Probleme cu colecţiile de obiecte . . . 192
7.4.2 Colecţii generice . . . 192
7.5 Elemente specifice C# 3.0 . . . 193
7.5.1 Proprietăţi implementate automat . . . 193
7.5.2 Iniţializatori de obiecte . . . 194
7.5.3 Iniţializatori de colecţii . . . 196

8 ADO.NET 199
8.1 Ce reprezintă ADO.NET? . . . 199
8.2 Furnizori de date în ADO.NET . . . 200
8.3 Componentele unui furnizor de date . . . 200
8.3.1 Clasele *Connection* . . . 201
8.3.2 Clasele *Command* . . . 201
8.3.3 Clasele *DataReader* . . . 202
8.3.4 Clasele *DataAdapter* . . . 202
8.3.5 Clasa *DataSet* . . . 202
8.4 Obiecte *Connection* . . . 202
8.4.1 Proprietăţi . . . 203
8.4.2 Metode . . . 205
8.4.3 Evenimente . . . 205
8.4.4 Stocarea stringului de conexiune în fişier de configurare 205
8.4.5 Gruparea conexiunilor . . . 207
8.4.6 Mod de lucru cu conexiunile . . . 207
8.5 Obiecte *Command* . . . 208
8.5.1 Proprietăţi . . . 208
8.5.2 Metode . . . 209
8.5.3 Utilizarea unei comenzi cu o procedură stocată . . . 212

8.5.4 Folosirea comenzilor parametrizate . . . 212
8.6 Obiecte *DataReader* . . . 214
8.6.1 Proprietăţi . . . 215
8.6.2 Metode . . . 215
8.6.3 Crearea şi utilizarea unui *DataReader* . . . 216
8.6.4 Utilizarea de seturi de date multiple . . . 217
8.6.5 Seturi de date cu tip . . . 217

9 ADO.NET (2) 219
9.1 Obiecte *DataAdapter* . . . 219
9.1.1 Metode . . . 220
9.1.2 Proprietăţi . . . 221
9.2 Clasa *DataSet* . . . 221
9.2.1 Conţinut . . . 222
9.2.2 Clasa *DataTable* . . . 222
9.2.3 Relaţii între tabele . . . 225
9.2.4 Popularea unui *DataSet* . . . 225
9.2.5 Clasa *DataTableReader* . . . 226
9.2.6 Propagarea modificărilor către baza de date . . . 227
9.3 Tranzacţii în ADO.NET . . . 229
9.4 Lucrul generic cu furnizori de date . . . 231
9.5 Tipuri nulabile . . . 233

10 LINQ (I) 235
10.1 Generalităţi . . . 235
10.2 Motivaţie . . . 237
10.2.1 Codul clasic ADO.NET . . . 237
10.2.2 Nepotrivirea de paradigme . . . 238
10.3 LINQ to Objects: exemplificare . . . 239
10.4 Mecanisme utilizate de LINQ . . . 240
10.4.1 Inferenţa tipului . . . 241
10.4.2 Tipuri anonime . . . 242
10.4.3 Metode parţiale . . . 242
10.4.4 Metode de extensie . . . 243
10.4.5 Expresii lambda . . . 245
10.5 Operatori LINQ . . . 245
10.6 LINQ to Objects . . . 248
10.6.1 Filtarea cu `Where` . . . 250
10.6.2 Operatorul de proiecţie . . . 251
10.6.3 Operatorul `SelectMany` . . . 252
10.6.4 Joncţiuni . . . 253

10.6.5 Grupare . . . 254
10.6.6 Ordonare . . . 254
10.6.7 Agregare . . . 256
10.6.8 Partiţionare . . . 257
10.6.9 Concatenarea . . . 258
10.6.10 Referirea de elemente din secvenţe . . . 258

11 LINQ (II): Linq to SQL 261
11.1 Obţinerea unui context de date . . . 261
11.2 Adăugarea, modificarea şi ştergerea de înregistrări în tabelă . 264
11.2.1 Adăugarea . . . 264
11.2.2 Modificarea unei înregistrări . . . 265
11.2.3 Ştergerea unei înregistrări . . . 265
11.3 Opţiuni de încărcare a datelor . . . 266

12 Atribute. Fire de execuţie 269
12.1 Atribute . . . 269
12.1.1 Generalităţi . . . 269
12.1.2 Atribute predefinite . . . 271
12.1.3 Exemplificarea altor atribute predefinite . . . 273
12.1.4 Atribute definite de utilizator . . . 276
12.2 Fire de execuţie . . . 279
12.3 Managementul thread–urilor . . . 279
12.3.1 Pornirea thread–urilor . . . 279
12.3.2 Metoda *Join()* . . . 282
12.3.3 Suspendarea firelor de execuţie . . . 282
12.3.4 Omorârea thread–urilor . . . 283
12.3.5 Sugerarea priorităţilor firelor de execuţie . . . 286
12.3.6 Fire în fundal şi fire în prim-plan . . . 287
12.4 Sincronizarea . . . 288
12.4.1 Clasa *Interlocked* . . . 291
12.4.2 Instrucţiunea *lock* . . . 292
12.4.3 Clasa *Monitor* . . . 293

13 Noutăţi în C# 4.0 297
13.1 Parallel Linq . . . 297
13.2 Parametri cu nume şi parametri opţionali . . . 299
13.3 Tipuri de date dinamice . . . 301
13.4 COM Interop . . . 302
13.5 Covarianţă şi contravarianţă . . . 304

14 Fluxuri **307**
14.1 Sistemul de fişiere . . . 307
14.1.1 Lucrul cu directoarele: clasele *Directory* şi *DirectoryInfo* 308
14.1.2 Lucrul cu fişierele: clasele *FileInfo* şi *File* . . . 310
14.2 Citirea şi scrierea datelor . . . 315
14.2.1 Clasa *Stream* . . . 315
14.2.2 Clasa *FileStream* . . . 317
14.2.3 Clasa *MemoryStream* . . . 317
14.2.4 Clasa *BufferedStream* . . . 318
14.2.5 Clasele *BinaryReader* şi *BinaryWriter* . . . 319
14.2.6 Clasele *TextReader*, *TextWriter* şi descendentele lor . . . 320
14.3 Operare sincronă şi asincronă . . . 322
14.4 Stream–uri Web . . . 325
14.5 Serializarea . . . 325
14.5.1 Crearea unui obiect serializabil . . . 325
14.5.2 Serializarea . . . 326
14.5.3 Deserializarea unui obiect . . . 326
14.5.4 Date tranziente . . . 327
14.5.5 Operaţii la deserializare . . . 327
Bibliografie . . . 329

Curs 1

Platforma Microsoft .NET

1.1 Prezentare generală

Platforma .NET 3.5 este un cadru de dezvoltare a softului, sub care se vor realiza, distribui si rula aplicaţiile de tip forme Windows, aplicaţii WEB si servicii WEB. Ea constă în trei părţi principale: Common Language Runtime, clasele specifice platformei şi ASP.NET. O infrastructură ajutătoare, Microsoft .NET Compact Framework este un set de interfeţe de programare care permite dezvoltatorilor realizarea de aplicaţii pentru dispozitive mobile precum telefoane inteligente şi PDA-uri[1].

.NET Framework constituie un nivel de abstractizare între aplicaţie şi nucleului sistemului de operare (sau alte programe), pentru a asigura portabilitatea codului; de asemenea integrează tehnologii care au fost lansate de catre Microsoft incepând cu mijlocul anilor 90 (COM, DCOM, ActiveX, etc) sau tehnologii actuale (servicii Web, XML).

Platforma constă în câteva grupe de produse:

1. *Unelte de dezvoltare* - un set de limbaje (C#, Visual Basic .NET, J#, C++/CLI, JScript.NET, Objective-C, Python, Smalltalk, Eiffel, Perl, Fortran, Cobol, Lisp, Haskell, Pascal, RPG, etc), un set de medii de dezvoltare (Visual Studio .NET, Visio), infrastructura .NET Framework, o bibliotecă cuprinzătoare de clase pentru crearea serviciilor Web (Web Services)[2], aplicaţiilor Web (Web Forms, ASP.NET MVC) şi a aplicaţiilor Windows (Windows Forms).

2. *Servere specializate* - un set de servere Enterprise .NET: SQL Server 2008, Exchange 2007, BizTalk Server 2006, SharePoint, etc, care pun la

[1] Definiţia oficială Microsoft, martie 2005, www.microsoft.com/net/basics/glossary.asp
[2] Serviciilor Web - aplicaţii care oferă servicii folosind Web-ul ca modalitate de acces.

dispoziţie funcţionalităţi diverse pentru stocarea bazelor de date, email, aplicaţii B2B[3].

3. *Servicii Web* - cel mai notabil exemplu este .NET Passport - un mod prin care utilizatorii se pot autentifica pe site-urile Web vizitate, folosind un singur nume şi o parolă pentru toate. Alt exemplu este MapPoint care permite vizualizarea, editarea si integrarea hartilor.

4. *Dispozitive* - noi dispozitive non–PC, programabile prin .NET Compact Framework, o versiune redusă a lui .NET Framework: Pocket PC Phone Edition, Smartphone, Tablet PC, Smart Display, XBox, set-top boxes, etc.

Motivul pentru care Microsoft a trecut la dezvoltarea acestei platforme este maturizarea industriei software, accentuându–se următoarele direcţii:

1. *Aplicaţiile distribuite* - sunt din ce în ce mai numeroase aplicaţiile de tip client / server sau cele pe mai multe nivele ($n-$tier). Tehnologiile distribuite actuale cer de multe ori o mare afinitate faţă de producător şi prezintă o carenţă acută a interoperării cu Web-ul. Viziunea actuală se depărtează de cea de tip client/server către una în care calculatoare, dispozitive inteligente şi servicii conlucrează pentru atingerea scopurilor propuse. Toate acestea se fac deja folosind standarde Internet neproprietare (HTTP, XML, SOAP).

2. *Dezvoltarea orientată pe componente* - este de mult timp cerută simplificarea integrării componentelor software dezvoltate de diferiţi producători. COM (Component Object Model) a realizat acest deziderat, dar dezvoltarea şi distribuirea aplicaţiilor COM este prea complexă. Microsoft .NET pune la dispoziţie un mod mai simplu de a dezvolta şi a distribui componente.

3. *Modificări ale paradigmei Web* - de-a lungul timpului s–au adus îmbunătăţiri tehnologiilor Web pentru a simplifica dezvoltarea aplicaţiilor. În ultimii ani, dezvoltarea aplicaţiilor Web s–a mutat de la prezentare (HTML şi adiacente) către capacitate sporită de programare (XML şi SOAP).

4. *Alţi factori de maturizare a industriei software* - reprezintă conştientizarea cererilor de interoperabilitate, scalabilitate, disponibilitate; unul din dezideratele .NET este de a oferi toate acestea.

[3]Bussiness to Bussiness - Comerţ electronic între parteneri de afaceri, diferită de comerţul electronic între client şi afacere (Bussiness to Consumer – B2C).

1.2 Arhitectura platformei Microsoft .NET

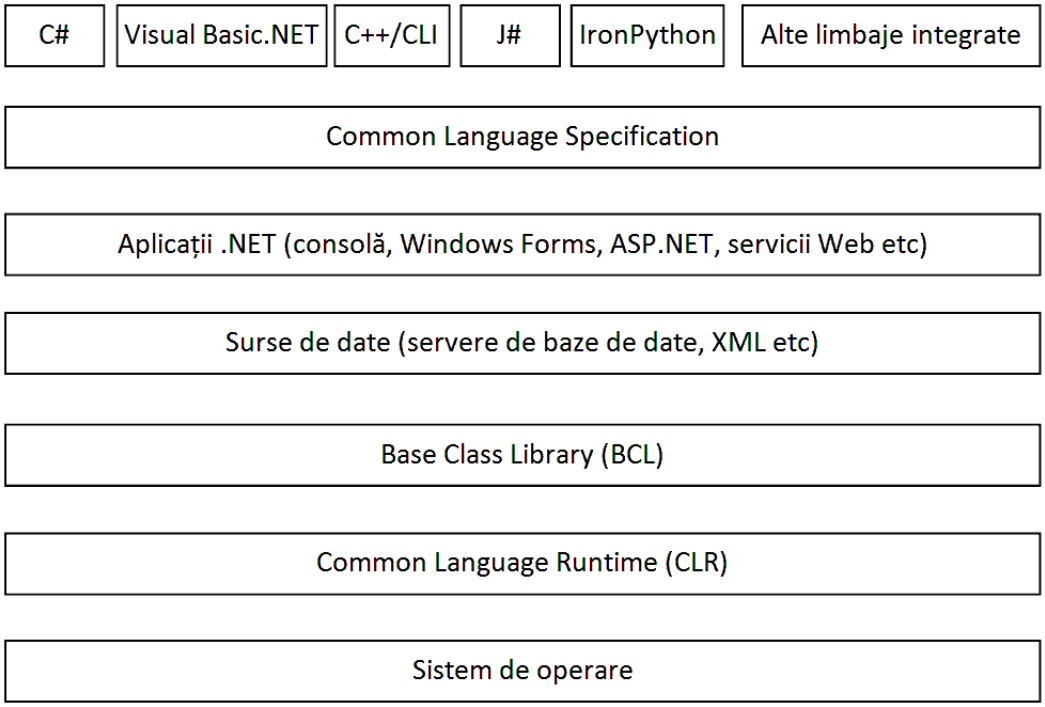

Figura 1.1: Arhitectura .NET

Figura 1.1 schematizează arhitectura platformei Microsoft .NET. Orice program scris într-unul din limbajele .NET este compilat în Common Intermediate Language[4], în concordanţă cu Common Language Specification (CLS). Aceste limbaje sunt sprijinite de o bogată colecţie de biblioteci de clase, ce pun la dispoziţie facilităţi pentru dezvoltarea de Web Forms, Windows Forms şi Web Services. Comunicarea dintre aplicaţii şi servicii se face pe baza unor clase de manipulare XML şi a datelor, ceea ce sprijină dezvoltarea aplicaţiilor cu arhitectură *n-tier*. Base Class Library există pentru a asigura funcţionalitate de nivel scăzut, precum operaţii de I/O, fire de execuţie, lucrul cu şiruri de caractere, comunicaţie prin reţea, etc. Aceste clase sunt reunite sub numele de .NET Framework Class Library, ce permite dezvoltarea rapidă a aplicaţiilor. La baza tuturor se află cea mai importantă componentă a lui .NET Framework - Common Language Runtime, care răspunde de execuţia fiecărui program. Evident, nivelul inferior este rezervat sistemului de operare. Trebuie spus că platforma .NET nu este exclusiv dezvoltată pentru sistemul

[4]Anterior numit şi Microsoft Intermediate Language (MSIL) sau Intermediate Language (IL).

de operare Microsoft Windows, ci şi pentru arome de Unix (FreeBSD sau Linux - a se vedea proiectul Mono[5]).

1.3 Componente ale lui .NET Framework

1.3.1 Common Intermediate Language

Una din uneltele de care dispune ingineria software este *abstractizarea*. Deseori vrem să ferim utilizatorul de detalii, să punem la dispoziţia altora mecansime sau cunoştinţe generale, care să permită atingerea scopului, fără a fi necesare cunoaşterea tuturor detaliilor. Dacă interfaţa rămâne neschimbată, se pot modifica toate detaliile interne, fără a afecta acţiunile celorlaţi beneficiari ai codului.

În cazul limbajelor de programare, s-a ajuns treptat la crearea unor nivele de abstractizare a codului rezultat la compilare, precum *p-code* (cel produs de compilatorul Pascal-P) şi *bytecode* (binecunoscut celor care au lucrat în Java). Bytecode-ul Java, generat prin compilarea unui fişier sursă, este cod scris într-un limbaj intermediar care suportă POO. Bytecod-ul este în acelaşi timp o abstractizare care permite executarea codului Java, indiferent de platforma ţintă, atâta timp cât această platformă are implementată o maşină virtuală Java, capabilă să "traducă" mai departe fişierul class în cod nativ.

Microsoft a realizat şi el propria sa abstractizare de limbaj, aceasta numindu-se Common Intermediate Language. Deşi există mai multe limbaje de programare de nivel înalt (C#, Managed C++, Visual Basic .NET, etc), la compilare toate vor produce cod în acelaşi limbaj intermediar: Common Intermediate Language. Asemănător cu bytecod-ul, CIL are trăsături OO, precum abstractizarea datelor, moştenirea, polimorfismul, sau concepte care s-au dovedit a fi extrem de necesare, precum excepţiile sau evenimentele. De remarcat că această abstractizare de limbaj permite rularea aplicaţiilor independent de platformă (cu aceeaşi condiţie ca la Java: să existe o maşină virtuală pentru acea platformă).

1.3.2 Common Language Specification

Unul din scopurile .NET este de a sprijini integrarea limbajelor astfel încât programele, deşi scrise în diferite limbaje, pot interopera, folosind din plin moştenirea, polimorfismul, încapsularea, excepţiile, etc. Dar limbajele nu sunt identice: de exemplu, unele sunt case sensitive, altele nu. Pentru a se asigura interoperabilitatea codului scris în diferite limbaje, Microsoft

[5] www.go-mono.com

a publicat Common Language Specification (CLS), un subset al lui CTS (Common Type System, vezi 1.3.4), conţinând specificaţii de reguli necesare pentru integrarea limbajelor.

CLS defineşte un set de reguli pentru compilatoarele .NET, asigurând faptul că fiecare compilator va genera cod care interferează cu platforma (mai exact, cu CLR–ul — vezi mai jos) într-un mod independent de limbajul sursă. Obiectele şi tipurile create în diferite limbaje pot interacţiona fără probleme suplimentare. Combinaţia CTS/CLS realizează de fapt interoperarea limbajelor. Concret, se poate ca o clasă scrisă în C# să fie moştenită de o clasă scrisă în Visual Basic care aruncă excepţii ce sunt prinse de cod scris în C++ sau J#.

1.3.3 Common Language Runtime

CLR este de departe cea mai importantă componentă a lui .NET Framework. Este responsabilă cu managementul şi execuţia codului scris în limbaje .NET, aflat în format CIL; este foarte similar cu Java Virtual Machine. CLR instanţiază obiectele, face verificări de securitate, depune obiectele în memorie, disponibilizează memoria prin garbage collection.

În urma compilării unei aplicaţii poate rezulta un fişier cu extensia exe, dar care nu este un executabil portabil Windows, ci un executabil portabil .NET (.NET PE). Acest cod nu este deci un executabil nativ, ci se va rula de către CLR, întocmai cum un fişier class este rulat în cadrul JVM. CLR foloseşte tehnologia compilării JIT - o implementare de maşină virtuală, în care o metodă sau o funcţie, în momentul în care este apelată pentru prima oară, este tradusă în cod maşină. Codul translatat este depus într-un cache, evitând-se astfel recompilarea ulterioară. Există 3 tipuri de compilatoare JIT:

1. *Normal JIT* - a se vedea descrierea de mai sus.
2. *Pre-JIT* - compilează întregul cod în cod nativ singură dată. În mod normal este folosit la instalări.
3. *Econo-JIT* - se foloşeşte pe dispozitive cu resurse limitate. Compilează codul CIL bit cu bit, eliberând resursele folosite de codul nativ ce este stocat în cache.

În esenţă, activitatea unui compilator JIT este destinată a îmbunătăţi performanţa execuţiei, ca alternativă la compilarea repetată a aceleiaşi bucăţi de cod în cazul unor apelări multiple. Unul din avantajele mecanismului JIT apare în clipa în care codul, o dată ce a fost compilat, se execută pe diverse

procesoare; dacă maşina virtuală este bine adaptată la noua platformă, atunci acest cod va beneficia de toate optimizările posibile, fără a mai fi nevoie recompilarea lui (precum în C++, de exemplu).

1.3.4 Common Type System

Pentru a asigura interoperabilitatea limbajelor din .NET Framework, o clasă scrisă în C# trebuie să fie echivalentă cu una scrisă în VB.NET, o interfaţă scrisă în Managed C++ trebuie să fie perfect utilizabilă în Managed Cobol. Toate limbajele care fac parte din pleiada .NET trebuie să aibe un set comun de concepte pentru a putea fi integrate. Modul în care acest deziderat s-a transformat în realitate se numeşte Common Type System (CTS); orice limbaj trebuie să recunoască şi să poată manipula nişte tipuri comune.

O scurtă descriere a unor facilităţi comune (ce vor fi exhaustiv enumerate şi tratate în cadrul prezentării limbajului C#):

1. *Tipuri valoare* - în general, CLR-ul (care se ocupă de managementul şi execuţia codului CIL, vezi mai jos) suportă două tipuri diferite: tipuri valoare şi tipuri referinţă. Tipurile valoare reprezintă tipuri alocate pe stivă şi nu pot avea valoare de null. Tipurile valoare includ tipurile primitive, structuri şi enumerări. Datorită faptului că de regulă au dimensiuni mici şi sunt alocate pe stivă, se manipuleaza eficient, reducând overhead-ul cerut de mecanismul de garbage collection.

2. *Tipuri referinţă* - se folosesc dacă variabilele de un anumit tip cer resurse de memorie semnificative. Variabilele de tip referinţă conţin adrese de memorie heap şi pot fi null. Transferul parametrilor se face rapid, dar referinţele induc un cost suplimentar datorită mecanismului de garbage collection.

3. *Boxing şi unboxing* - motivul pentru care există tipuri primitive este acelaşi ca şi în Java: performanţa. Însă orice variabilă în .NET este compatibilă cu clasa Object, rădăcina ierarhiei existente în .NET. De exemplu, *int* este un alias pentru *System.Int32*, care se derivează din *System.ValueType*. Tipurile valoare se stochează pe stivă, dar pot fi oricând convertite într-un tip referinţă memorat în heap; acest mecanism se numeşte *boxing*. De exemplu:

   ```
   int i = 1; //i - un tip valoare
   Object box = i; //box - un obiect referinta
   ```

   Când se face boxing, se obţine un obiect care poate fi gestionat la fel ca oricare altul, făcându–se abstracţie de originea lui.

Inversa boxing-ului este unboxing-ul, prin care se poate converti un obiect în tipul valoare echivalent, ca mai jos:

```
int j = (int)box;
```

unde operatorul de conversie este suficient pentru a converti de la un obiect la o variabilă de tip valoare.

4. *Clase, proprietăţi, indexatori* - platforma .NET suportă pe deplin programarea orientată pe obiecte, concepte legate de obiecte (încapsularea, moştenirea, polimorfismul) sau trăsături legate de clase (metode, câmpuri, membri statici, vizibilitate, accesibilitate, tipuri imbricate, etc). De asemenea se includ trăsături precum proprietăţi, indexatori, evenimente.

5. *Interfeţe* - reprezintă acelaşi concept precum clasele abstracte din C++ (dar conţinând doar funcţii virtuale pure), sau interfeţele Java. O clasă care se derivează dintr-o interfaţă trebuie să implementeze toate metodele acelei interfeţe. Se permite implementarea simultană a mai multor interfeţe (în rest moştenirea claselor este simplă).

6. *Delegaţi* - inspiraţi de pointerii la funcţii din C, ce permit programarea generică. Reprezintă versiunea "sigură" a pointerilor către funcţii din C/C++ şi sunt mecanismul prin care se tratează evenimentele.

Există numeroase componente ale lui CLR care îl fac cea mai importantă parte a lui .NET Framework: metadata, assemblies, assembly cache, reflection, garbage collection.

1.3.5 Metadate

Metadatele înseamnă "date despre date". În cazul .NET, ele reprezintă detalii destinate a fi citite şi folosite de către platformă. Sunt stocate împreună cu codul pe care îl descrie. Pe baza metadatelor, CLR ştie cum să instanţieze obiectele, cum să le apeleze metodele, cum să acceseze proprietăţile. Printr-un mecanism numit reflectare, o aplicaţie (nu neapărat CLR) poate să interogheze această metadată şi să afle ce expune un tip de date (clasă, structură, etc).

Mai pe larg, metadata conţine o declaraţie a fiecărui tip şi câte o declaraţie pentru fiecare metodă, câmp, proprietate, eveniment al tipului respectiv. Pentru fiecare metodă implementată, metadata conţine informaţie care permite încărcătorului clasei respective să localizeze corpul metodei. De asemena mai poate conţine declaraţii despre *cultura* aplicaţiei respective, adică despre localizarea ei (limba folosită în partea de interfaţă utilizator).

1.3.6 Assemblies

Un assembly reprezintă un bloc funcţional al unei aplicaţii .NET. El formează *unitatea fundamentală de distribuire, versionare, reutilizare şi permisiuni de securitate*. La runtime, un tip de date există în interiorul unui assembly (şi nu poate exista în exteriorul acestuia). Un assembly conţine metadate care sunt folosite de către CLR. Scopul acestor "assemblies" este să se asigure dezvoltarea softului în mod "plug-and-play". Dar metadatele nu sunt suficiente pentru acest lucru; mai sunt necesare şi "manifestele".

Un manifest reprezintă metadate despre assembly-ul care găzduieşte tipurile de date. Conţine numele assembly-ului, numărul de versiune, referiri la alte assemblies, o listă a tipurilor în assembly, permisiuni de securitate şi altele.

Un assembly care este împărţit între mai multe aplicaţii are de asemenea un *shared name*. Această informaţie care este unică este opţională, neapărând în manifestul unui assembly daca acesta nu a fost gândit ca o aplicaţie partajată.

Deoarece un assembly conţine date care îl descriu, instalarea lui poate fi făcută copiind assemblyul în directorul destinaţie dorit. Când se doreşte rularea unei aplicaţii conţinute în assembly, manifestul va instrui mediul .NET despre modulele care sunt conţinute în assembly. Sunt folosite de asemenea şi referinţele către orice assembly extern de care are nevoie aplicaţia.

Versionarea este un aspect deosebit de important pentru a se evita aşa–numitul "DLL Hell". Scenariile precedente erau de tipul: se instaleaza o aplicaţie care aduce nişte fişiere .dll necesare pentru functionare. Ulterior, o altă aplicaţie care se instalează suprascrie aceste fişiere (sau măcar unul din ele) cu o versiune mai nouă, dar cu care vechea aplicaţie nu mai funcţionează corespunzător. Reinstalarea vechii aplicaţii nu rezolvă problema, deoarece a doua aplicaţie nu va funcţiona. Deşi fişierele dll conţin informaţie relativ la versiune în interiorul lor, ea nu este folosită de către sistemul de operare, ceea ce duce la probleme. O soluţie la această dilemă ar fi instalarea fişierelor dll în directorul aplicaţiei, dar în acest mod ar dispărea reutilizarea acestor biblioteci.

1.3.7 Assembly cache

Assembly cache este un director aflat în mod normal în directorul %windir% \Assembly. Atunci când un assembly este instalat pe o maşină, el va fi adăugat în assembly cache. Dacă un assembly este descărcat de pe Internet, el va fi stocat în assembly cache, într-o zonă tranzientă. Aplicaţiile instalate vor avea assemblies într-un assembly cache global.

În acest assembly cache vor exista versiuni multiple ale aceluiaşi assembly. Dacă programul de instalare este scris corect, va evita suprascrierea assembly-urilor deja existente (şi care funcţionează perfect cu acplicaţiile instalate), adăugând doar noul assembly. Este un mod de rezolvare a problemei "DLL Hell", unde suprascrierea unei biblioteci dinamice cu o variantă mai nouă putea duce la nefuncţionarea corespunzătoare a aplicaţiilor anterior instalate. CLR este cel care decide, pe baza informaţiilor din manifest, care este versiunea corectă de assembly de care o aplicaţie are nevoie. Acest mecanism pune capăt unei epoci de tristă amintire pentru programatori şi utilizatori.

1.3.8 Garbage collection

Managementul memoriei este una din sarcinile cele mai consumatoare de timp în programare. Garbage collection este mecanismul care se declanşează atunci când alocatorul de memorie răspunde negativ la o cerere de alocare de memorie. Implementarea este de tip "mark and sweep": se presupune iniţial că toată memoria alocată se poate disponibiliza, dupa care se determină care din obiecte sunt referite de variabilele aplicaţiei; cele care nu mai sunt referite sunt dealocate, celelalte zone de memorie sunt compactate. Obiectele a căror dimensiune de memorie este mai mare decât un anumit prag nu mai sunt mutate, pentru a nu creşte semnificativ penalizarea de performanţă.

În general, CLR este cel care se ocupă de apelarea mecanismului de garbage collection. Totuşi, la dorinţă, programatorul poate sugera rularea lui.

1.4 Trăsături ale platformei .NET

Prin prisma celor prezentate pâna acum putem rezuma următoarele trăsături:

- **Dezvoltarea multilimbaj:** Deoarece există mai multe limbaje pentru această platformă, este mai uşor de implementat părţi specifice în limbajele cele mai adecvate. Numarul limbajelor curent implementate este mai mare decât 10. Această dezvoltare are în vedere şi debugging-ul aplicaţiilor dezvoltate în mai multe limbaje.

- **Independenţa de procesor şi de platformă:** CIL este independent de procesor. O dată scrisă şi compilată, orice aplicaţie .NET (al cărei management este făcut de către CLR) poate fi rulată pe orice platformă. Datorită CLR-ului, aplicaţia este izolată de particularităţile hardware sau ale sistemului de operare.

- **Managementul automat al memoriei:** Problemele de dealocare de memorie sunt în mare parte rezolvate; overhead-ul indus de către mecanismul de garbage collection este suportabil, fiind implementat în sisteme mult mai timpurii. Menţionăm totuşi că la ora actuală există încă posibilitatea de a avea memory leak intr-o aplicaţie gestionată de către platforma .NET[6].

- **Suportul pentru versionare:** Ca o lecţie învăţată din perioada de "DLL Hell", versionarea este acum un aspect de care se ţine cont. Dacă o aplicaţie a fost dezvoltată şi testată folosind anumite componente, instalarea unei componente de versiune mai nouă nu va atenta la buna funcţionare a aplicaţiei în discuţie: cele două versiuni vor coexista paşnic, alegerea lor fiind făcută pe baza manifestelor.

- **Sprijinirea standardelor deschise:** Nu toate dispozitivele rulează sisteme de operare Microsoft sau folosesc procesoare Intel. Din această cauză orice este strâns legat de acestea este evitat. Se foloseşte XML şi cel mai vizibil descendent al acestuia, SOAP. Deoarece SOAP este un protocol simplu, bazat pe XML, ce foloseşte ca protocol de transmisie HTTP[7], el poate trece uşor de firewall-uri, spre deosebire de DCOM sau CORBA.

- **Distribuirea uşoară:** Actualmente instalarea unei aplicaţii sub Windows înseamnă copierea unor fişiere în nişte directoare anume, modificarea unor valori în regiştri, instalare de componente COM, etc. Dezinstalarea completă a unei aplicaţii este in majoritatea cazurilor o utopie. Aplicaţiile .NET, datorită metadatelor şi reflectării trec de aceste probleme. Se doreşte ca instalarea unei aplicaţii să nu însemne mai mult decât copierea fişierelor necesare într-un director, iar dezinstalarea aplicaţiei să se facă prin ştergerea acelui director.

- **Arhitectură distribuită:** Noua filosofie este de a asigura accesul la servicii Web distribuite; acestea conlucrează la obţinerea informaţiei dorite. Platforma .NET asigură suport şi unelte pentru realizarea acestui tip de aplicaţii.

- **Interoperabilitate cu codul "unmanaged":** Codul "unmanaged" se referă la cod care nu se află în totalitate sub controlul CLR. El este rulat de CLR, dar nu beneficiază de CTS sau garbage collection. Este vorba de apelurile funcţiilor din DLL-uri, folosirea componentelor COM, sau

[6] How to Detect and Avoid Memory and Resource Leaks in .NET Applications
[7] Folosit la transmiterea paginilor Web pe Internet

folosirea de către o componentă COM a unei componente .NET. Codul existent se poate folosi în continuare.

- **Securitate:** Aplicaţiile bazate pe componente distribuite cer automat securitate. Modalitatea actuală de securizare, bazată pe drepturile contului utilizatorului, sau cel din Java, în care codul suspectat este rulat într-un "sandbox", fără acces la resursele critice este înlocuit în .NET de un control mai fin, pe baza metadatelor din assembly (zona din care provine - ex. Internet, intranet, maşina locală, etc) precum şi a politicilor de securitate ce se pot seta.

Curs 2

Vedere generală asupra limbajului C#. Tipuri predefinite. Tablouri. Şiruri de caractere

2.1 Vedere generală asupra limbajului C#

C#[1] este un limbaj de programare imperativ, obiect–orientat. Este foarte asemănător cu Java şi C++, motiv pentru care curba de învăţare este foarte lină.

Un prim exemplu de program, care conţine o serie de elemente ce se vor întâlni în continuare, este:

```
using System;
class HelloWorld
{
  public static void Main()
  {
    Console.WriteLine(‘‘Hello world!’’);
  }
}
```

Prima linie *using System;* este o directivă care specifică faptul că se vor folosi clasele care sunt incluse în spaţiul de nume[2] *System*; un spaţiu de nume este o colecţie de tipuri sau o grupare de alte spaţii de nume care pot fi folosite într-un program (detalii vor fi date mai târziu). În cazul de faţă, clasa care este folosită din acest spaţiu de nume este *Console*. Mai departe,

[1] "#" se pronunţă "sharp"

[2] Eng: namespace

orice program este conţinut într–o clasă, în cazul nostru *HelloWorld*. Punctul de intrare în aplicaţie este metoda *Main*, care nu preia în acest exemplu nici un argument din linia de comandă şi nu returnează explicit un indicator de stare a terminării.

Pentru operaţiile de intrare–ieşire cu consola se foloseşte clasa *Console*; pentru ea se apelează metoda statică *WriteLine*, care afişează pe ecran mesajul, după care face trecerea la linie nouă.

O variantă uşor modificată a programului de mai sus, în care se salută persoanele ale căror nume este transmis prin linia de comandă:

```
using System;
class HelloWorld
{
  public static void Main( String[] args)
  {
    for( int i=0; i<args.Length; i++)
    {
      Console.WriteLine( ‘‘Hello {0}’’, args[i]);
    }
  }
}
```

În exemplul precedent metoda principală preia o listă de parametri transmişi din linia de comandă (un şir de obiecte de tip *String*) şi va afişa pentru fiecare nume “Hello” urmat de numele de indice i (numerotarea parametrilor începe de la 0). Construcţia “{0}” va fi înlocuită cu primul argument care urmează după “Hello {0}”. La executarea programului de mai sus în forma: `HelloWorld Ana Dan`, se va afişa pe ecran:

```
Hello Ana
Hello Dan
```

Metoda Main poate să returneze o valoare întreagă, care să fie folosită de către sistemul de operare pentru a semnala dacă procesul s–a încheiat cu succes sau nu[3]. Menţionăm faptul că limbajul este case–sensitive[4].

Ca metode de notare Microsoft recomandă folosirea următoarelor două convenţii:

- conventie Pascal, în care prima literă a fiecărui cuvânt se scrie ca literă mare; exemplu: LoadData, SaveLogFile

[3]De exemplu în fişiere de comenzi prin testarea variabilei de mediu ERRORLEVEL
[4]Face distincţie între litere mari şi mici

- convenţia tip "cămilă" este la fel ca precedenta, dar primul caracter al primului cuvânt nu este scris ca literă mare; exemplu: userIdentifier, firstName.

În general, convenţia tip Pascal este folosită pentru tot ce este vizibil (public), precum nume de clase, metode, proprietăţi, etc. Parametrii metodelor şi numele câmpurilor se scriu cu convenţia cămilă. Se recomandă evitarea folosirii notaţiei ungare (numele unei entităţi trebuie să se refere la semantica ei, nu la tipul de reprezentare).

2.2 Tipuri de date

C# prezintă două grupuri de tipuri de date: *tipuri valoare* şi *tipuri referinţă*. Tipurile valoare includ tipurile simple (ex. *char, int, float*)[5], tipurile enumerare şi structură şi au ca principale caracteristici faptul că ele conţin direct datele referite şi sunt alocate pe stivă sau *inline* într–o structură. Tipurile referinţă includ tipurile clasă, interfaţă, delegat şi tablou, toate având proprietatea că variabilele de acest tip stochează referinţe către obiectele conţinute. Demn de remarcat este că toate tipurile de date sunt derivate (direct sau nu) din tipul System.Object, punând astfel la dispoziţie un mod unitar de tratare a lor.

2.2.1 Tipuri predefinite

C# conţine un set de tipuri predefinite, pentru care nu este necesară referirea vreunui spaţiu de nume via directiva `using` sau calificare completă: string, object, tipurile întregi cu semn şi fără semn, tipuri numerice în virgulă mobilă, tipurile bool şi decimal.

Tipul string este folosit pentru manipularea şirurilor de caractere codificate Unicode; conţinutul obiectelor de tip `string` nu se poate modifica[6]. Clasa object este rădăcina ierarhiei de clase din .NET, la care orice tip (inclusiv un tip valoare) poate fi convertit.

Tipul bool este folosit pentru a reprezenta valorile logice true şi false. Tipul char este folosit pentru a reprezenta caractere Unicode, reprezentate pe 16 biţi. Tipul decimal este folosit pentru calcule în care erorile determinate de reprezentarea în virgulă mobilă sunt inacceptabile, el punând la dispoziţie 28 de cifre zecimale semnificative.

[5] De fapt acestea sunt structuri, prezentate în alt curs

[6] Spunem despre un string că este *invariabil* - engl. *immutable*

Tabelul de mai jos conţine lista tipurilor predefinite, arătând totodată cum se scriu valorile corespunzătoare:

Tabelul 2.1: Tipuri predefinite.

Tip	Descriere	Exemplu
object	rădacina oricărui tip	object a = null;
string	o secvenţă de caractere Unicode	string s = “hello”;
sbyte	tip întreg cu semn, pe 8 biţi	sbyte val = 12;
short	tip întreg cu semn, pe 16 biţi	short val = 12;
int	tip întreg cu semn, pe 16 biţi	int val = 12;
long	tip întreg cu semn, pe 64 biţi	long val1 = 12; long val2=34L;
byte	tip întreg fără semn, pe 8 biţi	byte val = 12;
ushort	tip întreg fără semn, pe 16 biţi	ushort val = 12;
uint	tip întreg fără semn, pe 32 biţi	uint val = 12;

ulong	tip întreg fără semn, pe 64 de biţi	ulong val1=12;
		ulong val2=34U;
		ulong val3=56L;
		ulong val4=76UL;

float	tip cu virgulă mobilă, simplă precizie	float val=1.23F;
double	tip în virgulă mobilă, dublă precizie	double val1=1.23; double val2=4.56D;
bool	tip boolean	bool val1=false; bool val2=true;
char	tip caracter din setul Unicode	char val=’h’;
decimal	tip zecimal cu 28 de cifre semnificative	decimal val=1.23M;

Fiecare din tipurile predefinite este un alias pentru un tip pus la dispoziţie de sistem. De exemplu, string este alias pentru clasa System.String, int este alias pentru System.Int32.

2.2.2 Tipuri valoare

C# pune programatorului la dispoziţie tipuri valoare, care sunt fie structuri, fie enumerări. Există un set predefinit de structuri numite *tipuri simple*, identificate prin cuvinte rezervate. Un tip simplu este fie de tip numeric[7], fie boolean. Tipurile numerice sunt tipuri întregi, în virgulă mobilă sau decimal. Tipurile intregi sunt sbyte, byte, short, ushort, int, uint, long, ulong, char; cele în virgulă mobilă sunt float şi double. Tipurile enumerare se pot defini de către utilizator.

[7]Engl: integral type

Toate tipurile valoare derivează din clasa System.ValueType, care la rândul ei este derivată din clasa object (alias pentru System.Object). Nu este posibil ca dintr–un tip valoare să se deriveze. Atribuirea pentru un astfel de tip înseamnă copierea valorii dintr–o parte în alta.

Structuri

Un tip structură este un tip valoare care poate să conţină declaraţii de constante, câmpuri, metode, proprietăţi, indexatori, operatori, constructori sau tipuri imbricate. Vor fi descrise într–un capitol următor.

Tipuri simple

C# pune are predefinit un set de tipuri structuri numite tipuri simple. Tipurile simple sunt identificate prin cuvinte rezervate, dar acestea reprezintă doar alias–uri pentru tipurile struct corespunzătoare din spaţiul de nume System; corespondenţa este dată în tabelul de mai jos:

Tabelul 2.2: Tipuri simple şi corespondenţele lor cu tipurile din spaţiul de nume System.

Tabelul 2.2

Cuvânt rezervat	Tipul alias
sbyte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

Deoarece un tip simplu este un alias pentru un tip struct, orice tip simplu are membri. De exemplu, tipul int, fiind un tip alias pentru System.Int32, următoarele declaraţii sunt legale:

```
int i = int.MaxValue;    //constanta System.Int32.MaxValue
string s = i.ToString(); //metoda System.Int32.ToString()
string t = 3.ToString(); //idem
double d = Double.Parse("3.14");
```

Tipuri întregi

C# suportă nouă tipuri întregi: sbyte, byte, short, ushort, int, uint, long, ulong şi char. Acestea au următoarele dimensiuni şi domeniu de valori:

- sbyte reprezintă tip cu semn pe 8 biţi, cu valori de la -128 la 127;
- byte reprezintă tip fără semn pe 8 biţi, între 0 şi 255;
- short reprezintă tip cu semn pe 16 biţi, între -32768 şi 32767;
- ushort reprezintă tip fără semn pe 16 biţi, între 0 şi 65535;
- int reprezintă tip cu semn pe 32 de biţi, între -2^{31} şi $2^{31} - 1$;
- uint reprezintă tip fără semn pe 32 de biţi, între 0 şi $2^{32} - 1$;
- long reprezintă tip cu semn pe 64 de biţi, între -2^{63} şi $2^{63} - 1$;
- ulong reprezintă tip fără semn pe 64 de biţi, între 0 şi $2^{64} - 1$;
- char reprezintă tip fără semn pe 16 biţi, cu valori între 0 şi 65535. Mulţimea valorilor posibile pentru char corespunde setului de caractere Unicode.

Reprezentarea unei variable de tip întreg se poate face sub formă de şir de cifre zecimale sau hexazecimale, urmate eventual de un prefix. Numerele exprimate în hexazecimal sunt prefixate cu "0x" sau "0X". Regulile după care se asignează un tip pentru o valoare sunt:

1. dacă şirul de cifre nu are un sufix, atunci el este considerat ca fiind primul tip care poate să conţină valoarea dată: int, uint, long, ulong;
2. dacă şirul de cifre are sufixul u sau U, el este considerat ca fiind din primul tip care poate să conţină valoarea dată: uint, ulong;
3. dacă şirul de cifre are sufixul l sau L, el este considerat ca fiind din primul tip care poate să conţină valoarea dată: long, ulong;
4. dacă şirul de cifre are sufixul ul, uL, Ul, UL, lu, lU, Lu, LU, el este considerat ca fiind din tipul ulong.

Dacă o valoare este în afara domeniului lui ulong, apare o eroare la compilare.

Literalii de tip caracter au forma: 'caracter' unde "caracter" poate fi exprimat printr–un caracter, printr–o secvenţă escape simplă, secvenţă escape hexazecimală sau secvenţă escape Unicode. În prima formă poate fi folosit orice caracter exceptând apostrof, backslash şi new line. Secvenţă escape simplă poate fi: `\'`, `\"`, `\\`, `\0`, `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`, cu semnificaţiile cunoscute din C++. O secvenţă escape hexazecimală începe cu `\x` urmat de 1–4 cifre hexa. Deşi ca reprezentare, char este identic cu ushort, nu toate operaţiile ce de pot efectua cu ushort sunt valabile şi pentru char.

În cazul în care o operaţie aritmetică produce un rezultat care nu poate fi reprezentat în tipul destinaţie, comportamentul depinde de utilizarea operatorilor sau a declaraţiilor *checked* şi *unchecked* (care se pot utiliza în sursă sau din linia de compilare): în context checked, o eroare de depăşire duce la aruncarea unei excepţii de tip System.OverflowException. În context unchecked, eroarea de depăşire este ignorată, iar biţii semnificativi care nu mai încap în reprezentare sunt eliminaţi.

Exemplu:

```
byte i=255;
unchecked
{
  i++;
}//i va avea valoarea 0, nu se semnaleaza eroare
checked
{
  i=255;
  i++;//se va arunca exceptie System.OverflowException
}
```

Pentru expresiile aritmetice care conţin operatorii ++, --, +, - (unar şi binar), *, / şi care nu sunt conţinute în interiorul unui bloc de tip checked, comportamentul este specificat prin intermediul opţiunii /checked[+|−] dat din linia de comandă pentru compilator. Dacă nu se specifică nimic, atunci se va considera implicit unchecked.

Tipuri în virgulă mobilă

Sunt prezente 2 tipuri numerice în virgulă mobilă: float şi double. Tipurile sunt reprezentate folosind precizie de 32, respectivi 64 de biţi, folosind formatul IEC 60559, care permit reprezentarea valorilor de "0 pozitiv" şi "0 negativ" (se comportă la fel, dar anumite operaţii duc la obţinerea acestor

două valori), $+\infty$ şi $-\infty$ (obţinute prin împărţirea unui număr strict pozitiv, respectiv strict negativ la 0), a valorii Not–a–Number (NaN) (obţinută prin operaţii în virgulă mobilă invalide, de exemplu $0/0$ sau $\sqrt{-1}$), precum şi un set finit de numere. Tipul float poate reprezenta valori cuprinse între 1.5×10^{-45} şi 3.4×10^{38} (şi din domeniul negativ corespunzător), cu o precizie de 7 cifre. Double poate reprezenta valori cuprinse între 5.0×10^{-324} şi 1.7×10^{308} cu o precizie de 15-16 cifre.

Operaţiile cu floating point nu duc niciodată la apariţia de excepţii, dar ele pot duce, în caz de operaţii invalide, la valori 0, infinit sau NaN.

Literalii care specifică un număr reprezentat în virgulă mobilă au forma:
literal–real::
cifre-zecimale . cifre-zecimale exponent$_{optional}$ sufix-de-tip-real$_{optional}$
. cifre-zecimale exponent$_{optional}$ sufix-de-tip-real$_{optional}$
cifre-zecimale exponent sufix-de-tip-real$_{optional}$
cifre-zecimale sufix-de-tip-real,
unde
exponent::
e semn$_{optional}$ cifre-zecimale
E semn$_{optional}$ cifre-zecimale,
semn este + sau -, sufix-de-tip-real este F, f, D, d. Dacă nici un sufix de tip real nu este specificat, atunci literalul dat este de tip double. Sufixul f sau F specifică tip float, d sau D specifică double. Dacă literalul specificat nu poate fi reprezentat în tipul precizat, apare eroare de compilare.

Tipul decimal

Este un tip de date reprezentat pe 128 de biţi, gândit a fi folosit în calcule financiare sau care necesită precizie mai mare. Poate reprezenta valori aflate în intervalul 1.0×10^{-28} şi 7.9×10^{28}, cu 28 de cifre semnificative. Acest tip nu poate reprezenta zero cu semn, infinit sau NaN. Dacă în urma operaţiilor, un număr este prea mic pentru a putea fi reprezentat ca decimal, atunci el este făcut 0, iar dacă este prea mare, rezultă o excepţie. Diferenţa principală faţă de tipurile în virgulă mobilă este că are o precizie mai mare, dar un domeniu de reprezentare mai mic. Din cauza aceasta, nu se fac conversii implicite între nici un tip în virgulă mobilă şi decimal şi nu este posibilă mixarea variabilelor de acest tip într-o expresie, fără conversii explicite.

Literalii de acest tip se exprimă folosind ca sufix-de-tip-real caracterele m sau M. Dacă valoarea specificată nu poate fi reprezentată prin tipul decimal, apare o eroare la compilare.

Tipul bool

Este folosit pentru reprezentarea valorilor de adevăr true şi false. Literalii care se pot folosi sunt true şi false. Nu există conversii standard între bool şi nici un alt tip.

2.2.3 Tipul enumerare

Tipul enumerare este un tip valoare, construit pentru a permite declararea constantelor înrudite, într–o manieră clară şi sigură din punct de vedere al tipului. Un exemplu este:

```
using System;
public class Draw
{
  public enum LineStyle
  {
    Solid
    Dotted,
    DotDash
  }

  public void DrawLine(int x1, int y1, int x2, int y2,
    LineStyle lineStyle)
  {
    if (lineStyle == LineStyle.Solid)
    {
      //cod desenare linie continua
    }
    else
    if (lineStyle == LineStyle.Dotted)
    {
      //cod desenare linie punctata
    }
    else
    if (lineStyle == LineStyle.DotDash)
    {
      //cod desenare segment linie-punct
    }
    else
    {
      throw new ArgumentException(‘‘Invalid line style’’);
```

```
    }
  }
}

class Test
{
  public static void Main()
  {
    Draw draw = new Draw();
    draw.DrawLine(0, 0, 10, 10, Draw.LineStyle.Solid);
    draw.DrawLine(0, 0, 10, 10, (Draw.LineStyle)100);
  }
}
```

Al doilea apel este legal, deoarece valorile care se pot specifica pentru un enum nu sunt limitate la valorile declarate în enum. Ca atare, programatorul trebuie să facă validări suplimentare pentru a determina consistenţa valorilor. În cazul de faţă, la apelul de metodă se aruncă o excepţie (excepţiile vor fi tratate pe larg într-un curs viitor).

Ca şi mod de scriere a enumerărilor, se sugerează folosirea convenţiei Pascal atât pentru numele tipului cât şi pentru numele valorilor conţinute.

Enumerările nu pot fi declarate abstracte şi nu pot fi derivate. Orice enum este derivat automat din System.Enum, care este la rândul lui derivat din System.ValueType; astfel, metodele moştenite de la tipurile părinte sunt utilizabile de către orice variabilă de tip enum.

Fiecare tip enumerare care este folosit are un tip de reprezentare[8], pentru a se cunoaşte cât spaţiu de memorie trebuie să fie alocat unei variabile de acest tip. Dacă nu se specifică nici un tip de reprezentare (ca mai sus), atunci se presupune implicit tipul int. Specificarea unui tip de reprezentare (care poate fi orice tip integral, exceptând tipul char) se face prin enunţarea tipului după numele enumerării:

```
enum MyEnum : byte
{
  small,
  large
}
```

Specificarea este folosită atunci când dimensiunea în memorie este importantă, sau când se doreşte crearea unui tip de indicator (un tip flag) al cărui număr de stări diferă de numărul de biţi alocaţi tipului int (modelare de flag-uri):

[8]Engl: underlying type

```
enum ActionAttributes : ulong
{
Read = 1,
Write = 2,
Delete = 4,
Query = 8,
Sync = 16
//etc
}
...
ActionAttributes aa=ActionAttributes.Read|ActionAttributes.Write
| ActionAttributes.Query;
...
```

În mod implicit, valoarea primului membru al unei structuri este 0, şi fiecare variablă care urmează are valoarea mai mare cu o unitate decât precedenta. La dorinţă, valoarea fiecărui câmp poate fi specificat explicit:

```
enum Values
{
  a = 1,
  b = 2,
  c = a + b
}
```

Următoarele observaţii se impun relativ la lucrul cu tipul enumerare:

1. valorile specificate ca iniţializatori trebuie să fie reprezentabile prin tipul de reprezentare a enumerării, altfel apare o eroare la compilare:

   ```
   enum Out : byte
   {
     A = -1
   }//eroare semnalata la compilare
   ```

2. mai mulţi membri pot avea aceeaşi valoare (manevră dictată de semantica tipului construit):

   ```
   enum ExamState
   {
     passed = 10,
     failed = 1,
     rejected = failed
   }
   ```

3. dacă pentru un membru nu este dată o valoare, acesta va lua valoarea membrului precedent + 1 (cu excepţia primului membru – vezi mai sus)

4. nu se permit referinţe circulare:

```
enum CircularEnum
{
  A = B,
  B
}//A depinde explicit de B, B depinde implicit de A
//eroare semnalata la compilare
```

5. este recomandat ca orice tip enum să conţină un membru cu valoarea 0, pentru că în anumite contexte valoarea implicită pentru o variabilă enum este 0, ceea ce poate duce la inconsistenţe şi bug-uri greu de depanat

```
enum Months
{
  InvalidMonth,//are valoarea implicita 0, fiind primul element
  January,
  February,
  //etc
}
```

Tipurile enumerare pot fi convertite către tipul lor de bază şi înapoi, folosind o conversie explicită (cast):

```
enum Values
{
  a = 1,
  b = 5,
  c= 3
}

class Test
{
  public static void Main()
  {
    Values v = (Values)3;
    int ival = (int)v;
```

```
  }
}
```

Valoarea 0 poate fi convertită către un enum fără cast:

```
  ...
  MyEnum me;
  ...
  if (me == 0)
  {
    //cod
  }
```

Următorul cod arată câteva din artificiile care pot fi aplicate tipului enumerare: obţinerea tipului unui element de tip enumerare precum şi a tipului clasei de bază, a tipului de reprezentare, a valorilor conţinute (ca nume simbolice şi ca valori), conversie de la un string la un tip enumerare pe baza numelui, etc. Exemplul este preluat din <FrameworkSDK> \Samples\Technologies\ValueAn

```
using System;

namespace DemoEnum
{
class DemoEnum
{
enum Color
{
Red   = 111,
Green = 222,
Blue  = 333
}
private static void DemoEnums()
{
Console.WriteLine("\n\nDemo start: Demo of enumerated types.");
Color c = Color.Red;

// What type is this enum & what is it derived from
Console.WriteLine("   The " + c.GetType() + " type is derived from "
 + c.GetType().BaseType);

// What is the underlying type used for the Enum's value
Console.WriteLine("   Underlying type: " + Enum.GetUnderlyingType(
```

```
            typeof(Color)));

// Display the set of legal enum values
Color[] o = (Color[]) Enum.GetValues(c.GetType());
Console.WriteLine("\n   Number of valid enum values: " + o.Length);
for (int x = 0; x < o.Length; x++)
{
Color cc = ((Color)(o[x]));
Console.WriteLine("   {0}: Name={1,7}\t\tNumber={2}", x,
cc.ToString("G"), cc.ToString("D"));
}

// Check if a value is legal for this enum
Console.WriteLine("\n   111 is a valid enum value: " + Enum.IsDefined(
          c.GetType(), 111));   // True
Console.WriteLine("   112 is a valid enum value: " + Enum.IsDefined(
    c.GetType(), 112));   // False

// Check if two enums are equal
Console.WriteLine("\n   Is c equal to Red: " + (Color.Red == c));//True
Console.WriteLine("   Is c equal to Blue: "  + (Color.Blue == c));//False

// Display the enum's value as a string using different format specifiers
Console.WriteLine("\n   c's value as a string: " + c.ToString("G"));//Red
Console.WriteLine("   c's value as a number: " + c.ToString("D"));//111

// Convert a string to an enum's value
c = (Color) (Enum.Parse(typeof(Color), "Blue"));
try
{
c = (Color) (Enum.Parse(typeof(Color), "NotAColor"));//Not valid,
//raises exception
}
catch (ArgumentException)
{
Console.WriteLine("   'NotAColor' is not a valid value for this enum.");
}

// Display the enum's value as a string
Console.WriteLine("\n   c's value as a string: " + c.ToString("G"));//Blue
Console.WriteLine("   c's value as a number: " + c.ToString("D"));//333
```

```
Console.WriteLine("Demo stop: Demo of enumerated types.");
}
static void Main()
{
DemoEnums();
}
}
}
```

2.3 Tablouri

De multe ori se doreşte a se lucra cu o colecţie de elemente de un anumit tip. O soluţie pentru această problemă o reprezintă tablourile. Sintaxa de declarare este asemănătoare cu cea din Java sau C++, dar fiecare tablou este un obiect, derivat din clasa abstractă System.Array. Accesul la elemente se face prin intermediul indicilor care încep de la 0 şi se termină la numărul de elemente-1 (pentru un tablou unidimensional; în cadrul unui tablou multidimensional valoarea indicelui maxim este numărul de elemente de pe dimensiunea respectivă minus 1); orice depăşire a indicilor duce la apariţia unei excepţii: System.IndexOutOfRangeEx
O variabilă de tip tablou poate avea valoare de null sau poate să indice către o instanţă validă.

2.3.1 Tablouri unidimensionale

Declararea unui tablou unidimensional se face prin plasarea de paranteze drepte între numele tipului tabloului şi numele său, ca mai jos[9]:

```
int[] sir;
```

Declararea de mai sus nu duce la alocare de spaţiu pentru memorarea şirului; *instanţierea* se poate face ca mai jos:

```
sir = new int[10];
```

Exemplu:

```
using System;
class Unidimensional
{
```

[9]Spre deosebire de Java, nu se poate modifica locul parantezelor, adică nu se poate scrie: int sir[].

```
  public static int Main()
  {
    int[] sir;
    int n;
    Console.Write(‘‘Dimensiunea vectorului: ’’);
    n = Int32.Parse( Console.ReadLine() );
    sir = new int[n];
    for( int i=0; i<sir.Length; i++)
    {
      sir[i] = i * i;
    }
    for( int i=0; i<sir.Length; i++)
    {
      Console.WriteLine(‘‘sir[{0}]={1}’’, i, sir[i]);
    }
    return 0;
  }
}
```

În acest exemplu se foloseşte proprietatea[10] Length, care returnează numărul *tuturor* elementelor vectorului (lucru mai vizibil la tablourile multidimensionale rectangulare). De menţionat că în acest context n şi *sir* nu se pot declara la un loc, adică declaraţii de genul `int[] sir, n;` sau `int n, []sir;` sunt incorecte (prima este corectă din punct de vedere sintactic, dar ar rezulta că n este şi el un tablou; în al doilea caz, declaraţia nu este corectă sintactic).

Se pot face iniţializări ale valorilor conţinute într–un tablou:

```
int[] a = new int[] {1,2,3};
```

sau în forma mai scurtă:

```
int[] a = {1,2,3};
```

2.3.2 Tablouri multidimensionale

C# cunoaşte două tipuri de tablouri multidimensionale: rectangulare şi neregulate[11]. Numele lor vine de la forma pe care o pot avea.

[10] Pentru noţiunea de proprietate, vezi la partea despre clase.

[11] Engl: jagged arrays.

Tablouri rectangulare

Tablourile rectangulare au proprietatea că numărul de elemente pentru o anumită dimensiune este păstrat constant; altfel spus, acest tip de tablouri au chiar formă dreptunghiulară.

```
int[,] tab;
```

unde tab este un tablou rectangular bidimensional. Instanţierea se face:

```
tab = new int[2,3];
```

rezultând un tablou cu 2 linii şi 3 coloane; fiecare linie are exact 3 eleemnte si acest lucru nu se poate schimba pentru tabloul declarat. Referirea la elementul aflat pe linia i şi coloana j se face cu $tab[i, j]$.

La declararea tabloului se poate face şi iniţializare:

```
int[,] tab = new int[,] {{1,2},{3,4}};
```

sau, mai pe scurt:

```
int[,] tab = {{1, 2}, {3, 4}};
```

Exemplu:

```
using System;
class Test
{
  public static void Main()
  {
    int[,] tabInm = new int[10,10];
    for( int i=0; i<tabInm.GetLength(0); i++ )
    {
      for( int j=0; j<tabInm.GetLength(1); j++)
      {
        tabInm[i,j] = i * j;
      }
    }
    for( int i=0; i<tabInm.GetLength(0); i++)
    {
      for( int j=0; j<tabInm.GetLength(1); j++)
      {
        Console.WriteLine(``{0}*{1}={2}'', i, j, tabInm[i,j]);
      }
```

```
    }
    Console.WriteLine(‘‘tabInm.Length={0}’’, tabInm.Length);
  }
}
```

După ce se afişează tabla înmulţirii până la 10, se va afişa: `tabInm.Length=100`, deoarece proprietatea Length dă numărul total de elemente aflat în tablou (pe toate dimensiunile). Am folosit însă metoda `GetLength(d)` care returnează numărul de elemente aflate pe dimensiunea numărul d (numărarea dimensiunilor începe cu 0).

Determinarea numărului de dimensiuni pentru un tablou rectangular la run–time se face folosind proprietatea Rank a clasei de bază System.Array.

Exemplu:

```
using System;
class Dimensiuni
{
  public static void Main()
  {
    int[] t1 = new int[2];
    int[,] t2 = new int[3,4];
    int[,,] t3 = new int[5,6,7];
    Console.WriteLine(‘‘t1.Rank={0}\nt2.Rank={1}\nt3.Rank={2}’’,
    t1.Rank, t2.Rank, t3.Rank);
  }
}
```

Pe ecran va apărea:

```
t1.Rank=1
t2.Rank=2
t3.Rank=3
```

Tablouri neregulate

Un tablou neregulat[12] reprezintă un tablou de tabouri. Declararea unui tablou neregulat cu două dimensiuni se face ca mai jos:

```
int[][] tab;
```

Referirea la elementul de indici i şi j se face prin `tab[i][j]`.

Exemplu:

[12]Engl: jagged array

```
using System;
class JaggedArray
{
  public static void Main()
  {
    int[][] a = new int[2][];
    a[0] = new int[2];
    a[1] = new int[3];
    for( int i=0; i<a[0].Length; i++)
    {
      a[0][i] = i;
    }
    for( int i=0; i<a[1].Length; i++)
    {
      a[1][i] = i * i;
    }
    for(int i=0; i<a.Length; i++)
    {
      for( int j=0; j<a[i].Length; j++ )
      {
        Console.Write(``{0} '', a[i][j]);
      }
      Console.WriteLine();
    }
    Console.WriteLine(``a.Rank={0}'', a.Rank);
  }
}
```

va scrie pe ecran:

```
0 1
0 1 4
a.Rank=1
```

Ultima linie afişată se explică prin faptul că un tablou neregulat este un vector care conţine referinţe, deci este unidimensional.

Iniţializarea valorilor unui tablou neregulat se poate face la declarare:

```
int[][] myJaggedArray = new int [][]
                        {
                           new int[] {1,3,5,7,9},
                           new int[] {0,2,4,6},
```

```
                        new int[] {11,22}
                    };
```

Forma de mai sus se poate prescurta la:

```
int[][] myJaggedArray = {
                        new int[] {1,3,5,7,9},
                        new int[] {0,2,4,6},
                        new int[] {11,22}
                    };
```

2.4 Şiruri de caractere

Tipul de date folosit pentru reprezentarea şirurilor de caractere este clasa System.String (pentru care se poate folosi aliasul "string"; reamintim că este un tip predefinit). Obiectele de acest tip sunt imutabile (caracterele conţinute nu se pot schimba, dar pe baza unui şir se poate obţine un alt şir). Şirurile pot conţine secvenţe escape şi pot fi de două tipuri: regulate şi de tip "verbatim"[13]. Şirurile regulate sunt demarcate prin ghilimele şi necesită secvenţe escape pentru reprezentarea caracterelor escape.

Exemplu:

```
String a = "string literal";
String versuri = "vers1\nvers2";
String caleCompleta = "\\\\minimax\\protect\\csharp";
```

Pentru situaţia în care se utilizează masiv secvenţe escape, se pot folosi şirurile verbatim. Un literal de acest tip are simbolul "@" înaintea ghilimelelor de început. Pentru cazul în care ghilimelele sunt întâlnite în interiorul şirului, ele se vor dubla. Un şir de caractere poate fi reprezentat pe mai multe rânduri fără a folosi caracterul \n. Şirurile verbatim sunt folosite pentru a face referiri la fişiere sau chei în regiştri, sau pentru expresii regulate.

Exemple:

```
String caleCompleta=@"\\minimax\protect\csharp";
//ghilimelele se dubleaza intr-un verbatim string
String s=@"notiunea ""aleator"" se refera...";
//string multilinie reprezentat ca verbatim
String dialog=@"-Alo? Cu ce va ajutam?
-As avea nevoie de o informatie.";
```

[13] Engl: verbatim literals

Operatorii "==" şi "!=" pentru două şiruri de caractere se comportă în felul următor: două şiruri de caractere se consideră egale dacă sunt fie amândouă null, fie au aceeaşi lungime şi caracterele de pe aceleaşi poziţii coincid; "!=" dă negarea relaţiei de egalitate. Clasa String pune la dispoziţie metode pentru: comparare (Compare, CompareOrdinal, CompareTo), căutare (EndsWith, StartsWith, IndexOf, LastIndexOf), modificare (a se înţelege obţinerea altor obiecte pe baza celui curent - Concat, CopyTo, Insert, Join, PadLeft, PadRight, Remove, Replace, Split, Substring, ToLower, ToUpper, Trim, TrimEnd, TrimStart)[14]. Accesarea unui caracter aflat pe o poziţie i a unui şir s se face prin folosirea parantezelor drepte, cu aceleaşi restricţii asupra indicelui ca şi pentru tablouri: `s[i]`.

În clasa object se află metoda ToString() care este suprascrisă în fiecare clasă ale cărei instanţe pot fi tipărite. Pentru obţinerea unei reprezentări diferite se foloseşte metoda `String.Format()`.

Un exemplu de folosire a funcţiei `Split()` pentru despărţirea unui şir în funcţie de separatori este:

```
using System;
class Class1
{
  static void Main(string[] args)
  {
    String s = "Oh, I hadn't thought of that!";
    char[] x = {' ', ','};
    String[] tokens = s.Split( x );
    for(int i=0; i<tokens.Length; i++)
    {
      Console.WriteLine("Token: {0}", tokens[i]);
    }
  }
}
```

va afişa pe ecran:

```
Token: Oh
Token:
Token: I
Token: hadn't
Token: thought
Token: of
Token: that!
```

[14] A se vedea exemplele din MSDN.

De remarcat că pentru caracterul apostrof nu este obligatorie secvenţa escape în cazul şirurilor de caractere. Al doilea lucru care trebuie explicat este că al doilea token este cuvântul vid, care apare între cei doi separatori alăturaţi: virgula şi spaţiul. Metoda Split() nu face gruparea mai multor separatori, lucru care ar fi de dorit în prezenţa a doi separatori alăturaţi. Pentru aceasta putem apela la două metode. Prima presupune folosirea unei variante supraîncărcate a metodei Split, în care se precizează ca al doilea parametru optiunea de ignorare a rezultatelor goale:

```
String[] tokens = s.Split( new char[]{' ', ','},
     StringSplitOptions.RemoveEmptyEntries );
```

A doua modalitate se bazează pe folosirea expresiilor regulate.

Pentru a lucra cu şiruri de caractere care permit modificarea lor (concatenări repetate, substituiri de subşiruri) se foloseşte clasa StringBuilder, din spaţiul de nume System.Text.

2.4.1 Expresii regulate

În cazul în care funcţiile din clasa String nu sunt suficient de puternice, namespace–ul System.Text.RegularExpresions pune la dispoziţie o clasă de lucru cu expresii regulate numită Regex. Expresiile regulate reprezintă o metodă extrem de facilă de a opera căutări/înlocuiri pe text. Forma expresiilor regulate este cea din limbajul Perl.

Această clasă foloseşte o tehnică interesantă pentru mărirea performanţelor: dacă programatorul vrea, se scrie o secvenţă "din mers" pentru a implementa potrivirea expresiei regulate, după care codul este rulat[15].

Exemplul anterior poate fi rescris corect din puct de vedere al funcţionalităţii prin folosirea unei expresii regulate, pentru a prinde şi cazul separatorilor multipli adiacenţi:

```
class ExpresieRegulata
{
  static void Main(string[] args)
  {
    String s = "Oh, I hadn't thought of that!";
    //separator: virgula, spatiu sau punct si virgula
    //unul sau mai multe, orice combinatie
    Regex regex = new Regex("[, ;]+");
    String[] strs = regex.Split(s);
```

[15] Codul este scris direct în IL.

```
    for( int i=0; i<strs.Length; i++)
    {
      Console.WriteLine("Word: {0}", strs[i]);
    }
}
```

care va produce:

```
Word: Oh
Word: I
Word: hadn't
Word: thought
Word: of
Word: that!
```

Curs 3

Clase – generalităţi. Instrucţiuni. Spaţii de nume

3.1 Clase – vedere generală

Clasele reprezintă tipuri referinţă. O clasă poate să moştenească o singură clasă şi poate implementa mai multe interfeţe.

Clasele pot conţine constante, câmpuri, metode, proprietăţi, evenimente, indexatori, operatori, constructori de instanţă, destructori, constructori de clasă, tipuri imbricate. Fiecare membru poate conţine un nivel de protecţie, care controlează gradul de acces la el. O descriere este dată în tabelul 3.1:

Tabelul 3.1: Modificatori de acces ai membrilor unei clase

Accesor	**Semnificaţie**
public	Acces nelimitat
protected	Acces limitat la clasa conţinătoare sau la tipuri derivate din ea
internal	Acces limitat la acest assembly
protected internal	Acces limitat la acest assembly sau la tipuri derivate din clasă
private	Acces limitat la clasă; modificatorul implicit de acces

```
using System;
```

```
class MyClass
{
  public MyClass()
  {
    Console.WriteLine("Constructor instanta");
  }
  public MyClass( int value )
  {
    myField = value;
    Console.WriteLine("Constructor instanta");
  }
  public const int MyConst = 12;
  private int myField = 42;
  public void MyMethod()
  {
    Console.WriteLine("this.MyMethod");
  }
  public int MyProperty
  {
    get
    {
      return myField;
    }
    set
    {
      myField = value;
    }
  }
  public int this[int index]
  {
    get
    {
      return 0;
    }
    set
    {
      Console.WriteLine("this[{0}]={1}", index, value);
    }
  }
  public event EventHandler MyEvent;
  public static MyClass operator+(MyClass a, MyClass b)
```

```
  {
    return new MyClass(a.myField + b.myField);
  }
}

class Test
{
  static void Main()
  {
    MyClass a = new MyClass();
    MyClass b = new MyClass(1);
    Console.WriteLine("MyConst={0}", MyClass.MyConst);
    //a.myField++;//gradul de acces nu permite lucrul direct cu campul
    a.MyMethod();
    a.MyProperty++;
    Console.WriteLine("a.MyProperty={0}", a.MyProperty);
    a[3] = a[1] = a[2];
    Console.WriteLine("a[3]={0}", a[3]);
    a.MyEvent += new EventHandler(MyHandler);
    MyClass c = a + b;
  }

  static void MyHandler(object Sender, EventArgs e)
  {
    Console.WriteLine("Test.MyHandler");
  }

  internal class MyNestedType
  {}
}
```

Constanta este un membru al unei clase care reprezintă o valoare nemodificabilă, care poate fi evaluată la compilare. Constantele pot depinde de alte constante, atâta timp cât nu se creează dependenţe circulare. Ele sunt considerate automat membri statici (dar este interzis să se folosească specificatorul static în faţa lor). Ele pot fi accesate exclusiv prin intermediul numelui de clasă (MyClass.MyConst), şi nu prin intermediul vreunei instanţe (a.MyConst).

Câmpul este un membru asociat fiecărui obiect; câmpul stocheaza o valoare care contribuie la starea obiectului.

Metoda este un membru care implementează un calcul sau o acţiune care poate fi efectuată asupra unui obiect sau asupra unei clase. Metodele statice (care au în antet cuvântul cheie static) sunt accesate prin intermediul numelui de clasă, pe când cele nestatice (metode instanţă) sunt apelate prin intermediul unui obiect.

Proprietatea este un membru care dă acces la o caracteristică a unui obiect sau unei clase. Exemplele folosite până acum includeau lungimea unui vector, numărul de caractere ale unui şir de caractere, etc. Sintaxa pentru accesara câmpurilor şi a proprietăţilor este aceeaşi. Reprezintă o altă modalitate de implementare a accesorilor pentru obiecte.

Evenimentul este un membru care permite unei clase sau unui obiect să pună la dispoziţia altora notificări asupra evenimentelor. Tipul acestei declaraţii trebuie să fie un tip delegat. O instanţă a unui tip delegat încapsulează una sau mai multe entităţi apelabile. Exemplu:

```
public delegate void EventHandler(object sender,
                                  System.EventArgs e);
public class Button
{
  public event EventHandler Click;
  public void Reset()
  {
    Click = null;
  }
}

using System;
public class Form1
{
  Button Button1 = new Button1();

  public Form1()
  {
    Button1.Click += new EventHandler(Button1_Click);
  }

  void Button1_Click(object sender, EventArgs e )
  {
    Console.WriteLine("Button1 was clicked!");
```

```
  }

  public void Disconnect()
  {
    Button1.Click -= new EventHandler(Button1_Click);
  }
}
```

Mai sus clasa Form1 adaugă Button1_Click ca tratare de eveniment[1] pentru evenimentul Click al lui Button1. În metoda Disconnect(), acest event handler este înlăturat.

Operatorul este un membru care defineşte semnificaţia (supraîncărcarea) unui operator care se aplică instanţelor unei clase. Se pot supraîncărca operatorii binari, unari şi de conversie.

Indexatorul este un membru care permite unui obiect să fie indexat în acelaşi mod ca un tablou (pentru programatorii C++: supraîncărcarea operatorului []).

Constructorii instanţă sunt membri care implementează acţiuni cerute pentru iniţializarea fiecărui obiect.

Destructorul este un membru special care implementează acţiunile cerute pentru a distruge o instanţă a unei clase. Destructorul nu are parametri, nu poate avea modificatori de acces, nu poate fi apelat explicit şi este apelat automat de către garbage collector.

Constructorul static este un membru care implementează acţiuni necesare pentru a iniţializa o clasă, mai exact membrii statici ai clasei. Nu poate avea parametri, nu poate avea modificatori de acces, nu este apelat explicit, ci automat de către sistem.

Moştenirea este de tip simplu, iar rădăcina ierarhiei este clasa object (alias System.Object).

3.2 Transmiterea de parametri

În general, transmiterea parametrilor se face prin valoare. Acest lucru înseamnă că la apelul unei metode în stiva gestionată de compilator se copiază valoarea parametrului actual transmis, iar la revenire din metodă această

[1] Engl: event handler

valoare va fi ştearsă. Exemplificăm mai jos acest lucru pentru tipurile valoare şi referinţă.

```
using System;
class DemoTipValoare
{
  static void f(int x)
  {
    Console.WriteLine("la intrare in f: {0}", x );
    ++x;
    Console.WriteLine("la iesire din f: {0}", x );
  }
  static void Main()
  {
    int a = 100;
    Console.WriteLine("inainte de intrare in f: {0}", a);
    f( a );
    Console.WriteLine("dupa executarea lui f: {0}", a);
  }
}
```

Executarea acestui program va avea ca rezultat:

```
inainte de intrare in f: 100
la intrare in f: 100
la iesire din f: 101
dupa executarea lui f: 100
```

Pentru variable de tip referinţă, pe stivă se depune tot o copie a valorii obiectului. Însă pentru un asemenea tip de variabilă acest lucru înseamnă că pe stivă se va depune ca valoare adresa de memorie la care este stocat obiectul respectiv. Ca atare, metoda apelată poate să modifice starea obiectului care se transmite, dar nu obiectul in sine (adică referinţa sa):

```
class Employee
{
  public String name;
  public decimal salary;
}

class Test
{
```

```
  static void Main()
  {
    Employee e = new Employee();
    e.name = "Ionescu";
    e.salary = 300M;
    System.Console.WriteLine("pre: name={0}, salary={1}",
                        e.name, e.salary );
    Method( e );
    System.Console.WriteLine("post: name={0}, salary={1}",
                        e.name, e.salary );
  }

  static void Method( Employee e )
  {
    e.salary += 100;
  }
}
```

va avea ca rezultat:

```
pre: name=Ionescu, salary=300
post: name=Ionescu, salary=400
```

Totuşi, chiar şi în cazul tipului referinţă încercarea de a re–crea în interiorul unei metode un obiect transmis ca parametru nu are nici un efect după terminarea ei:

```
class MyClass
{
  public int x;
}

class Test
{
  static void f(MyClass myClass)
  {
    Console.WriteLine("intrare in f: {0}", myClass.x);
    myClass = new MyClass();
    myClass.x = -100;
    Console.WriteLine("iesire din f: {0}", myClass.x);
  }
  static void Main()
```

```
  {
    MyClass myClass = new MyClass();
    myClass.x = 100;
    Console.WriteLine("inainte de apel: {0}", myClass.x);
    f( myClass );
    Console.WriteLine("dupa apel: {0}", myClass.x);
}
}
```

Ieşirea acestui program va fi:

```
inainte de apel: 100
intrare in f: 100
iesire din f: -100
dupa apel: 100
```

Există situaţii în care acest comportament nu este cel dorit: am vrea ca efectul asupra unui parametru să se menţină şi după ce metoda apelată s–a terminat.

Un parametru *referinţă* este folosit tocmai pentru a rezolva problema transmiterii prin valoare, folosind referinţă (un alias) pentru entitatea dată de către metoda apelantă. Pentru a transmite un parametru prin referinţă, se prefixează cu cuvântul cheie ref la apel sau la declarare de metodă:

```
using System;
class Test
{
  static void Swap( ref int a, ref int b)
  {
    int t = a;
    a = b;
    b = t;
  }

  static void Main()
  {
    int x=1, y=2;
    Console.WriteLine("inainte de apel: x={0}, y={1}", x, y);
    Swap( ref a, ref b )
    Console.WriteLine("dupa apel: x={0}, y={1}", x, y);
  }
}
```

va realiza interschimbarea valorilor a şi b.

Una din trăsăturile specifice parametrilor referinţă este că valorile pentru care se face apelul trebuie să fie iniţializate. Neasignarea de valori pentru x şi y în exemplul de mai sus duce o eroare de compilare. Mai clar, exemplul de mai jos genereaza eroare la compilare, mesajul fiind: "Use of unassigned local variable 'x'":

```
class TestRef
{
  static void f(ref int x)
  {
    x = 100;
  }
  static void Main()
  {
    int x;
    f(ref x);
  }
}
```

Există cazuri în care dorim să obţinem acelaşi efect ca la parametrii referinţă, dar fără a trebui să iniţializăm argumentele date de către metoda apelantă (de exemplu când valoarea acestui parametru se calculează în interiorul metodei apelate). Pentru aceasta există parametrii de ieşire[2], similar cu parametrii referinţă, cu deosebirea că nu trebuie asignată o valoare parametrului de apel:

```
using System;
class Test
{
  static void Main()
  {
    int l = 10;
    double area;
    ComputeSquareArea( l, out area);
    Console.WriteLine("Area is: {0}", area);
  }

  static void ComputeSquareArea( double l, out double area )
  {
```

[2]Engl: output parameters

```
    area = l * l;
  }
}
```

Pentru toate tipurile de parametri de mai sus există o mapare 1 la 1 între parametrii actuali şi cei formali. Un parametru vector[3] permite o relaţie de tipul unul-la-mulţi: mai mulţi parametri actuali pot fi referite prin intermediul unui singur parametru formal. Un astfel de parametru se declară folosind modificatorul params. Pentru o implementare de metodă, putem avea cel mult un parametru de tip vector şi acesta trebuie să fie ultimul în lista de parametri. Acest parametru formal este tratat ca un tablou unidimensional:

```
using System;
class Test
{
  static void F(params int[] args)
  {
    Console.WriteLine("# of parameters: {0}", args.Length);
    for( int i=0; i<args.Length; i++)
    {
      Console.WriteLine("args[{0}]={1}", i, args[i]);
    }
  }

  static void Main()
  {
    F();
    F(1);
    F(1,2);
    F(new int[] {1,2,3});
  }
}
```

Acest tip de transmitere se foloseşte şi de către metoda WriteLine (sau Write) a clasei Console, i.e. există în această clasă o metodă de tipul:

```
public static void WriteLine(string format, params object[] args){...}
```

[3]Engl: parameter array

3.3 Conversii

O conversie permite ca o expresie de un anumit tip să fie tratată ca fiind de alt tip. Conversiile pot fi implicite sau explicite, aceasta specificând de fapt dacă un operator de conversie este sau nu necesar.

3.3.1 Conversii implicite

Sunt clasificate ca şi conversii implicite următoarele:

- conversiile identitate
- conversiile numerice implicite
- conversiile implicite de tip enumerare
- conversiile implicite de referinţe
- boxing
- conversiile implicite ale expresiilor constante
- conversii implicite definite de utilizator

Conversiile implicite pot apărea într–o varietate de situaţii, de exemplu apeluri de funcţii sau atribuiri. Conversiile implicite predefinite nu determină niciodată apariţia de excepţii.

Conversiile indentitate

O conversie identitate converteşte de la un tip oarecare către acelaşi tip.

Conversiile numerice implicite

Conversiile numerice implicite sunt:

- de la sbyte la short, int, long, float, double, decimal;
- de la byte la short, ushort, int, uint, long, ulong, float, double, decimal;
- de la short la int, long, double, decimal;
- de la ushort la int, uint, long, ulong, float, double, decimal;
- de la int la long, float, double, decimal;

- de la uint la long, ulong, float, double, decimal;
- de la long la float, double, decimal;
- de la ulong la float, double, decimal;
- de la char la ushort, int, uint, long, ulong, float, double, decimal;
- de la float la double.

Conversiile de la int, uint, long, ulong la float, precum şi cele de la long sau ulong la double pot duce la o pierdere a preciziei, dar niciodată la o reducere a ordinului de mărime. Alte conversii numerice implicite niciodată nu duc la pierdere de informaţie.

Conversiile de tip enumerare implicite

O astfel de conversie permite ca literalul 0 să fie convertit la orice tip enumerare (chiar dacă acesta nu conţine valoarea 0) - a se vedea 2.2.3, pag. 32.

Conversii implicite de referinţe

Conversiile implicite de referinţe implicite sunt:

- de la orice tip referinţă la object;
- de la orice tip clasă B la orice tip clasă A, dacă B este derivat din A;
- de la orice tip clasă A la orice interfaţă B, dacă A implementează B;
- de al orice interfaţă A la orice interfaţă B, dacă A este derivată din B;
- de la orice tip tablou A cu tipul A_E la un tip tablou B având tipul B_E, cu următoarele condiţii:
  1. A şi B au acelaşi număr de dimensiuni;
  2. atât A_E cât şi B_E sunt tipuri referinţă;
  3. există o conversie implicită de tip referinţă de la A_E la B_E
- de la un tablou la System.Array;
- de la tip delegat la System.Delegate;
- de la orice tip tablou sau tip delegat la System.ICloneable;
- de la tipul null la orice variabilă de tip referinţă;

Conversie de tip boxing

Permite unui tip valoare să fie implicit convertit către tipul object sau System.ValueType sau către orice tip interfaţă pe care tipul valoare îl implementează. O descriere mai amănunţită este dată în secţiunea 3.3.4.

3.3.2 Conversiile implicite ale expresiilor constante

Permit următoarele tipuri de conversii:

- o expresie constantă de tip int poate fi convertită către tipurile sbyte, byte, short, ushort, uint, ulong, cu condiţia ca valoarea expresiei constante să se afle în domeniul tipului destinaţie;
- o expresie constantă de tip long poate fi convertită la tipul ulong, dacă valoarea ce se converteşte nu este negativă.

Conversii implicite definite de utilizator

Constau într–o conversie implicită standard opţională, urmată de execuţia unui operator de conversie implicită utilizator urmată de altă conversie implicită standard opţională. Regulile exacte sunt descrise în [6].

3.3.3 Conversii explicite

Următoarele conversii sunt clasificate ca explicite:

- toate conversiile implicite
- conversiile numerice explicite
- conversiile explicite de enumerări
- conversiile explicite de referinţe
- unboxing
- conversii explicite definite de utilizator

Din cauză că orice conversie implicită este de asemenea şi una explicită, aplicarea operatorului de conversie este redundantă:

```
int x = 0;
long y = (long)x;//(long) este redundant
```

Conversii numerice explicite

Sunt conversii de la orice tip numeric la un alt tip numeric pentru care nu există conversie numerică implicită:

- de la sbyte la byte, ushort, uint, ulong, char;
- de la byte la sbyte, char;
- de la short la sbyte, byte, ushort, uint, ulong, char;
- de la ushort la sbyte, byte, short, char;
- de la int la sbyte, byte, short, ushort, int, char;
- de la uint la sbyte, byte, short, ushort, int, uint, long, ulong, char;
- de la long la sbyte, byte, short, ushort, int, uint, ulong, char;
- de la ulong la sbyte, byte, short, ushort, int, uint, long, char;
- de la char la sbyte, byte, short;
- de la float la sbyte, byte, short, ushort, int, uint, long, ulong, decimal;
- de la double la sbyte, byte, short, ushort, int, uint, long, ulong, char, float, decimal;
- de la decimal la sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double;

Pentru că în astfel de conversii pot apărea pierderi de informaţie, există două contexte în care se fac aceste conversii: checked şi unchecked.

În context checked, conversia se face cu succes dacă valoarea care se converteşte este reprezentabilă de către tipul către care se face conversia. În cazul în care conversia nu se poate face cu succes, se va arunca excepţia System.OverflowException. În context unchecked, conversia se face întotdeauna, dar se poate ajunge la pierdere de informaţie sau la valori ce nu sunt bine nedefinite (vezi [6], pag. 115–116).

Conversii explicite de enumerări

Conversiile explicite de enumerări sunt:

- de la sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal la orice tip enumerare;

- de la orice tip enumerare la sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal;
- de la orice tip enumerare la orice tip enumerare.

Conversiile de tip enumerare se fac prin tratarea fiecărui tip enumerare ca fiind tipul întreg de reprezentare, după care se efectuează o conversie implictă sau explicită între tipuri (ex: dacă se doreşte conversia de la un tip enumerare E care are tipul de reprezentare int la un tip byte, se va face o conversie explicită de la int la byte; invers, se va face o conversie implictă de la byte la int).

Conversii explicite de referinţe

Conversiile explicite de referinţe sunt:

- de la object la orice tip referinţă;
- de la orice tip clasă A la orice tip clasă B, cu condiţia ca A să fie clasă de bază pentru B;
- de la orice tip clasă A la orice tip interfaţă B, dacă A nu este nederivabilă şi A nu implementează pe B;
- de la orice tip interfaţă A la orice tip clasă B, dacă B nu este nederivabilă sau cu condiţia ca B să implementeze A;
- de la orice tip interfaţă A la orice tip interfaţă B, dacă A nu este derivat din B;
- de la un tip tablou A cu elemente de tip A_E la un tip tablou B cu elemente B_E, cu condiţiile:
  1. A şi B au acelaşi număr de dimensiuni;
  2. A_E şi B_E sunt tipuri referinţă;
  3. există o conversie de referinţă explicită de la A_E al B_E
- de la System.Array şi interfeţele pe care le implementează la orice tip tablou;
- de la System.Delegate şi interfeţele pe care le implementează la orice tip delegat.

Acest tip de conversii cer verificare la run–time. Dacă o astfel de conversie eşuează, se va arunca o excepţie de tipul System.InvalidCastException.

Unboxing

Unboxing-ul permite o conversie explicită de la object sau System.ValueType la orice tip valoare, sau de la orice tip interfaţă la orice tip valoare care implementează tipul interfaţă. Mai multe detalii se vor da în secţiunea 3.3.4.

Conversii explicite definite de utilizator

Constau într–o conversie standard explicită opţională, urmată de execuţia unei conversii explicite, urmată de o altă conversie standard explicită opţională.

3.3.4 Boxing şi unboxing

Boxing–ul şi unboxing–ul reprezintă modalitatea prin care C# permite utilizarea simplă a sistemului unificat de tipuri. Spre deosebire de Java, unde există tipuri primitive (care nu pot conţine metode) şi tipuri referinţă, în C# toate tipurile sunt derivate din clasa object (alias System.Object). De exemplu, tipul int (alias System.Int32) este derivat din clasa System.ValueType care la rândul ei este derivată din clasa object (alias System.Object). Ca atare, un întreg este compatibil cu object.

Boxing

Conversia de tip boxing permite oricărui tip valoare să fie implicit convertit către tipul object sau către un tip interfaţă implementat de tipul valoare. Boxing–ul unei valori constă în alocarea unei variabile de tip obiect şi copierea valorii iniţiale în acea instanţă.

Procesul de boxing al unei valori sau variabile de tip valoare se poate înţelege ca o simulare de creare de clasă pentru acel tip:

```
sealed class T_Box
{
  T value;
  public T_Box(T t)
  {
    value = t;
  }
}
```

Astfel, declaraţiile:

```
int i = 123;
object box = i;
```

corespund conceptual la:

```
int i = 123;
object box = new int_Box(i);
```

Pentru secvenţa:

```
int i = 10;//linia 1
object o = i;//linia 2
int j = (int)o;//linia 3
```

procesul se desfăşoară ca în figura 3.1: la linia 1, se declară şi se iniţializează o variabilă de tip valoare, care va conţine valoarea 10.La următoarea linie se va crea o referinţă o către un obiect alocat în heap, care va conţine atât valoarea 10, cât şi o informaţie despre tipul de dată conţinut (în cazul nostru, System.Int32). Unboxing–ul se face printr–o convenţie explicită, ca în linia a treia.

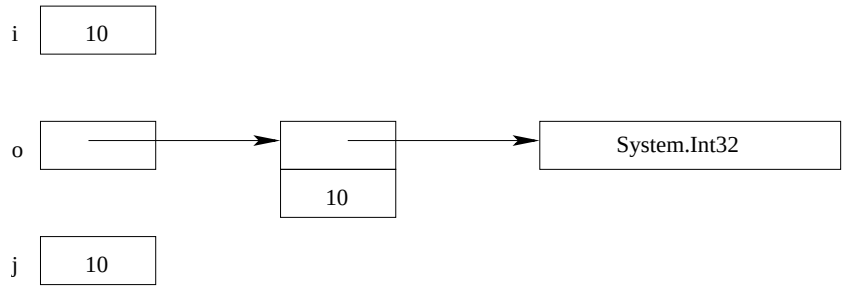

Figura 3.1: Boxing şi unboxing

Determinarea tipului pentru care s–a făcut împachetarea se face prin intermediul operatorului *is*:

```
int i = 123;
object o = i;
if (o is int)
{
  Console.Write("Este un int inauntru!");
}
```

Boxing–ul duce la o clonare a valorii care va fi conţinută. Altfel spus, secvenţa:

```
int i = 10;
object o = i;
i++;
Console.WriteLine("in o: {0}", o);
```

va afişa valoarea înglobată în obiect, 10.

3.4 Declaraţii de variabile şi constante

Variabilele şi constantele trebuie declarate în C#. Opţional, pentru variabile se poate specifica valoarea iniţială, iar pentru constante acest lucru este obligatoriu. O variabilă trebuie să aibă valoarea asignată definită înainte ca valoarea ei să fie utilizată, în cazul în care este declarată în interiorul unei metode. Este o eroare ca într–un sub-bloc să se declare o variabilă cu acelaşi nume ca în blocul conţinător:

```
void F()
{
  int x = 3, y;//ok
  const double d = 1.1;//ok
  {
    string x = "Mesaj: ";//eroare, x mai este declarat
    //in blocul continator
    int z = x + y;//eroare, y nu are o valoare definita asignata
  }
}
```

Constantele au valori iniţiale care trebuie să se poată evalua la compilare.

3.5 Instrucţiuni C#

3.5.1 Declaraţii de etichete

O etichetă poate prefixa o instrucţiune. Ea este vizibilă în întregul bloc şi toate sub-blocurile conţinute. O etichetă poate fi referită de către o instrucţiune *goto*:

```
class DemoLabel
{
  int F(int x)
  {
    if (x >= 0) goto myLabel;
    x = -x;
    myLabel: return x;
  }
  static void Main()
  {
    DemoLabel dl = new DemoLabel();
    dl.f();
```

```
  }
}
```

3.5.2 Instrucţiuni de selecţie

Instrucţiunea if

Instrucţiunea if execută o instrucţiune în funcţie de valoarea de adevăr a unei expresii logice. Are formele:

```
if (expresie logica) instructiune;
if (expresie logica) instructiune; else instructiune;
```

Instrucţiunea switch

Permite executarea unei instrucţiuni în funcţie de valoarea unei expresii, care se poate regăsi sau nu într–o listă de valori candidat:

```
switch (expresie)
{
  case eticheta: instructiune;
  case eticheta: instructiune;
  ...
  default: instructiune;
}
```

O etichetă reprezintă o expresie constantă. O instrucţiune poate să şi lipsească şi în acest caz se va executa instrucţiunea de la *case*–ul următor, sau de la *default*. Secţiunea *default* poate să lipsească. Dacă o instrucţiune este nevidă, atunci va trebui să fie terminată cu o instrucţiune *break* sau *goto case expresieConstanta* sau *goto default*.

Expresia după care se face selecţia poate fi de tip *sbyte, byte, short, ushort, int, uint, long, ulong, char, string, enumerare*. Dacă valoarea expresiei se regăseşte printre valorile specificate la clauzele case, atunci instrucţiunea corespunzătoare va fi executată; dacă nu, atunci instrucţiunea de la clauza default va fi executată (dacă ea există). Spre deosebire de C şi C++, e interzis să se folosească fenomenul de "cădere" de la o etichetă la alta; continuarea se face folosind explicit *goto*.

```
switch (i)
{
  case 0:
         Console.WriteLine("0");
```

```
        break;
  case 1:
       Console.Write("Valoarea ");
       goto case 2;
  case 2:
  case 3:
        Console.WriteLine(i);
        break;
  case 4:
        goto default;
  default:
        Console.WriteLine("Numar in afara domeniului admis");
      break;//neaparat, altfel eroare de compilare
}
```

Remarcăm în exemplul de mai sus că chiar şi în cazul lui *default* e necesar să se folosească instrucţiune de salt (în cazul nostru *break*); o motivaţie ar fi că această clauză *default* nu e necesar să fie trecută ultima în *switch*, ci chiar şi pe prima poziţie – desigur caz mai rar întâlnit.

Există un caz în care *break*, *goto case valoare* sau *goto default* pot să lipsească: când este evident că o asemenea instrucţiune *break/goto* nu ar putea fi atinsă (i.e. sunt prezente instrucţiunile *return*, *throw* sau o ciclare despre care se poate afirma la compilare că este infinită).

3.5.3 Instrucţiuni de ciclare

Există 4 instrucţiuni de ciclare: while, do, for, foreach.

Instrucţiunea while

Permite executarea unei instrucţiuni atâta timp cât valoarea unei expresii logice este adevărată (ciclu cu test anterior).

Sintaxa:

```
while (expresie logica) instructiune;
```

În interiorul unei astfel de instrucţiuni se poate folosi o instrucţiune de salt de tip *break* sau *continue*.

```
while (r != 0)
{
  r = a%b;
  a = b;
```

```
  b = r;
}
```

Instrucţiunea do

Execută o instrucţiune o dată sau de mai multe ori, cât timp o condiţie logică este adevărată (ciclu cu test posterior).

Exemplu:

```
do
{
  S += i++;
}while(i<=n)
```

Poate conţine instrucţiuni *break* sau *continue*.

Instrucţiunea for

Execută o secvenţă de iniţializare, după care va executa o instrucţiune atâta timp cât o condiţie este adevărată (ciclu cu test anterior); poate să conţină un pas de reiniţializare (trecerea la pasul următor). Se permite folosirea instrucţiunilor *break* şi *continue*.

Exemplu:

```
for (int i=0; i<n; i++)
{
  Console.WriteLine("i={0}", i);
}
```

Instrucţiunea foreach

Enumeră elementele dintr–o coleţie, executând o instrucţiune pentru fiecare element. Colecţia poate să fie orice instanţă a unei clase care implementează interfaţa *System.Collections.IEnumerable*.

Exemplu:

```
int[] t = {1, 2, 3};
foreach( int x in  t)
{
  Console.WriteLine(x);
}
```

Elementul care se extrage este de tip read–only (deci nu poate fi transmis ca parametru *ref* sau *out* şi nu se poate aplica un operator care să îi schimbe valoarea).

3.5.4 Instrucţiuni de salt

Permit schimbarea ordinii de execuţie a instrucţiunilor. Ele sunt: *break, continue, goto, return, throw.*

Instrucţiunea break

Produce ieşirea forţată dintr–un ciclu de tip *while, do, for, foreach.*

Instrucţiunea continue

Porneşte o nouă iteraţie în interiorul celui mai apropiat ciclu conţinător de tip *while, do, for, foreach.*

Instrucţiunea goto

Goto permite saltul al o anumită instrucţiune. Are 3 forme:

```
goto eticheta;
goto case expresieconstanta;
goto default;
```

Cerinţa este ca eticheta la care se face saltul să fie definită în cadrul funcţiei curente şi saltul să nu se facă în interiorul unor blocuri de instrucţiuni, deoarece nu se poate reface întotdeauna contextul acelui bloc.

Se recomandă evitarea utilizării intense a acestui cuvânt cheie, în caz contrar se poate ajunge la fenomenul de "spagetti code". Pentru o argumentare consistentă a acestei indicaţii, a se vedea articolul clasic al lui Edsger W. Dijkstra, "Go To Statement Considered Harmful": http://www.acm.org/classics/oct95/

Instrucţiunea return

Determină cedarea controlului funţiei apelante de către funcţia apelată. Dacă funcţia apelată are tip de retur, atunci instrucţiunea return trebuie să fie urmată de o expresie care suportă o conversie implicită către tipul de retur.

3.5.5 Instrucţiunile try, throw, catch, finally

Permit tratarea excepţiilor. Vor fi studiate în detaliu la capitolul de excepţii.

3.5.6 Instrucţiunile checked şi unchecked

Controlează contextul de verificare de depăşire a domeniului pentru aritmetica pe întregi şi conversii. Au forma:

```
checked
{
  //instructiuni
}
unchecked
{
  //instructiuni
}
```

Verificare se va face la run–time.

3.5.7 Instrucţiunea lock

Obţine excluderea mutuală asupra unui obiect pentru executarea unui bloc de instrucţiuni. Are forma:

```
lock (x) instructiune
```

X trebuie să fie de tip referinţă (dacă este de tip valoare, nu se face boxing).

3.5.8 Instrucţiunea using

Determină obţinerea a unei sau mai multor resurse, execută o instrucţiune şi apoi disponibilizează resursa:

```
using ( achizitie de resurse ) instructiune
```

O resursă este o clasă sau o structură care implementează interfaţa *System.IDisposable*, care include o sigură metodă fără parametri *Dispose()*. Achiziţia de resurse se poate face sub formă de variabile locale sau a unor expresii; toate acestea trebuie să fie implicit convertibile la *IDisposable*. Variabilele locale alocate ca resurse sunt read–only. Resursele sunt automat dealocate (prin apelul de *Dispose*) la sfârşitul instrucţiunii (care poate fi bloc de instrucţiuni).

Motivul pentru care există această instrucţiune este unul simplu: uneori se doreşte ca pentru anumite obiecte care deţin resurse importante să se apeleze automat metodă *Dispose()* de dezalocare a lor, cât mai repede cu putinţă.

Exemplu:

```
using System;
using System.IO;
class Test
{
  static void Main()
  {
    using( TextWriter w = File.CreateText("log.txt") )
    {
      w.WriteLine("This is line 1");
      w.EriteLine("This is line 2");
    }
  }
}
```

3.6 Spaţii de nume

În cazul creării de tipuri este posibil să se folosească un acelaşi nume pentru tipurile noi create de către dezvoltatorii de soft. Pentru a putea folosi astfel de clase care au numele comun, dar responsabilităţi diferite, trebuie prevăzută o modalitate de a le adresa în mod unic. Soluţia la această problemă este crearea spaţiilor de nume[4] care rezolvă, printr–o adresare completă astfel de ambiguităţi. Astfel, putem folosi de exemplu clasa Buffer din spaţiul System (calificare completă: System.Buffer), alături de clasa Buffer din spaţiul de nume Curs3: Curs3.Buffer.

Crearea unui spaţiu de nume se face prin folosirea cuvântului namespace:

```
using System;
namespace Curs3
{
  public class Buffer
  {
    public Buffer()
    {
      Console.WriteLine("Bufferul meu!");
    }
  }
}
```

Se pot de asemenea crea spaţii de nume imbricate. Altfel spus, un spaţiu de nume este o colecţie de tipuri sau de alte spaţii de nume.

[4] engl: namespaces

3.6.1 Declaraţii de spaţii de nume

O declaraţie de spaţiu de nume constă în cuvântul cheie *namespace*, urmat de identificatorul spaţiului de nume şi de blocul spaţiului de nume, delimitat de acolade. Spaţiile de nume sunt implicit publice şi acest tip de acces nu se poate modifica. În interiorul unui spaţiu de nume se pot utiliza alte spaţii de nume, pentru a se evita calificarea completă a claselor.

Identificatorul unui spaţiu de nume poate fi simplu sau o secvenţă de identificatori separaţi prin ".". Cea de a doua formă permite definirea de spaţii de nume imbricate, fără a se imbrica efectiv:

```
namespace N1.N2
{
  class A{}
  class B{}
}
```

este echivalentă cu:

```
namespace N1
{
  namespace N2
  {
    class A{}
    class B{}
  }
}
```

Două declaraţii de spaţii de nume cu aceeaşi denumire contribuie la declararea unui acelaşi spaţiu de nume:

```
namespace N1.N2
{
  class A{}
}

namespace N1.N2
{
  class B{}
}
```

este echivalentă cu cele două declaraţii anterioare.

3.6.2 Directiva using

Directiva using facilitează în primul rând utilizarea spaţiilor de nume şi a tipurilor definite în acestea; ele nu creează membri noi în cadrul unităţii de program în care sunt folosite, ci au rol de a uşura referirea tipurilor. Nu se pot utiliza în interiorul claselor, structurilor, enumerăririlor.

Exemplu: e mai uşor de înteles un cod de forma:

```
using System;
class A
{
  static void Main()
  {
    Console.WriteLine("Mesaj");
  }
}
```

decât:

```
class A
{
  static void Main()
  {
    System.Console.WriteLine("Mesaj");
  }
}
```

Directiva using poate fi de fapt folosită atât pentru importuri simbolice, cât şi pentru crearea de aliasuri.

Directiva using pentru import simbolic

O directivă using permite importarea simbolică a tuturor tipurilor conţinute direct într–un spaţiu de nume, i.e. folosirea lor fără a fi necesară o calificare completă. Acest import nu se referă şi la spaţiile de nume conţinute:

```
namespace N1.N2
{
  class A{}
}
namespace N3.N4
{
  class B{};
```

```
}
namespace N5
{
  using N1.N2;
  using N3;
  class C
  {
    A a = null;//ok
    N4.B = null;//Eroare, N4 nu a fost importat
  }
}
```

Importarea de spaţii de nume nu trebuie să ducă la ambiguităţi:

```
namespace N1
{
  class A{}
}
namespace N2
{
  class A{}
}
namespace N3
{
  using N1;
  using N2;
  class B
  {
    A a = null;//ambiguitate: N1.A sau N2.A?
  }
}
```

În situaţia de mai sus, conflictul (care poate foarte uşor în cazul în care se folosesc tipuri produse de dezvoltatori diferiţi) poate fi rezolvat de o calificare completă:

```
namespace N3
{
  using N1;
  using N2;
  class B
  {
```

```
    N1.A a1 = null;
    N2.A a2 = null;//ok, nu mai este ambiguitate
  }
}
```

Tipurile declarate în interiorul unui spaţiu de nume pot avea modificatori de acces *public* sau *internal* (modificatorul implicit). Un tip *internal* nu poate fi folosit prin import în afara assembly-ului, pe când unul *public*, da.

Directiva using ca alias

Introduce un identificator care serveşte drept alias pentru un spaţiu de nume sau pentru un tip.

Exemplu:

```
namespace N1.N2
{
  class A{}
}
namespace N3
{
  using A = N1.N2.A;
  class B
  {
     A a = null;
  }
}
```

Acelaşi efect se obţine creînd un alias la spaţiul de nume N1.N2:

```
namespace N3
{
  using N = N1.N2;
  class B
  {
     N.A a = null;
  }
}
```

Identificatorul dat unui alias trebuie să fie unic, adică în interiorul unui namespace nu trebuie să existe tipuri şi aliasuri cu acelaşi nume:

```
namespace N3
{
  class A{}
}
namespace N3
{
  using A = N1.N2.A;//eroare, deoarece simbolul A mai este definit
}
```

O directivă alias afectează doar blocul în care este definită:

```
namespace N3
{
  using R = N1.N2;
}
namespace N3
{
  class B
  {
    R.A a = null;//eroare, R nu este definit aici
  }
}
```

adică directiva de alias nu este tranzitivă. Situaţia de mai sus se poate rezolva prin declarearea aliasului în afara spaţiului de nume:

```
using R = N1.N2;
namespace N3
{
  class B
  {
    R.A a = null;
  }
}
namespace N3
{
  class C
  {
    R.A b = null;
  }
}
```

Numele create prin directive de alias sunt ascunse de către alte declaraţii care folosesc acelaşi identificator în interiorul unui bloc:

```
using R = N1.N2;
namespace N3
{
  class R{}
  class B
  {
   R.A a;//eroare, clasa R nu are membrul A
  }
}
```

Directivele de alias nu se influenţează reciproc:

```
namespace N1.N2{}
namespace N3
{
  using R1 = N1;
  using R2 = N1.N2;
  using R3 = R1.N2;//eroare, R1 necunoscut
}
```

3.7 Declararea unei clase

Declararea unei clase se face în felul următor:
atribute$_{opt}$ modificatori-de-clasa$_{opt}$ `class` identificator clasa-de-baza$_{opt}$ corp-clasa ;$_{opt}$

Modificatorii de clasă sunt:

public - clasele publice sunt accesibile de oriunde; poate fi folosit atât pentru clase imbricate, cât şi pentru clase care sunt conţinute în spaţii de nume;

internal - se poate folosi atât pentru clase imbricate, cât şi pentru clase care sunt conţinute în spaţii de nume (este modificatorul implicit pentru clase care sunt conţinute în spaţii de nume). Semnifică acces permis doar în clasa sau spaţiul de nume care o cuprinde;

protected - se poate specifica doar pentru clase imbricate; tipurile astfel calificate sunt accesibile în clasa curentă sau în cele derivate (chiar dacă clasa derivată face parte din alt spaţiu de nume);

private - doar pentru clase imbricate; semnifică acces limitat la clasa conţinătoare; este modificatorul implicit;

protected internal - folosibil doar pentru clase imbricate; tipul definit este accesibil în spaţiul de nume curent, în clasa conţinătoare sau în tipurile derivate din clasa conţinătoare;

new - permis pentru clasele imbricate; clasa astfel calificată ascunde un membru cu acelaşi nume care este moştenit;

sealed - o clasă sealed nu poate fi moştenită; poate fi clasă imbricată sau nu;

abstract - clasa care este incomplet definită şi care nu poate fi instanţiată; folosibilă pentru clase imbricat sau conţinute în spaţii de nume;

partial - clasa este definită în mai multe fişiere

3.8 Membrii unei clase

Corpul unei clase se specifică în felul următor:
{ declaratii-de-membri };$_{opt}$
Membrii unei clase sunt împărţiti în următoarele categorii:

- constante
- câmpuri
- metode
- proprietăţi
- evenimente
- indexatori
- operatori
- constructori (de instanţă)
- destructor
- constructor static
- tipuri

Acestor membri le pot fi ataşaţi modificatorii de acces:

public - membrul este accesibil de oriunde;

protected - membrul este accesabil de către orice membru al clasei conţinătoare şi de către clasele derivate;

internal - membrul este accesabil doar în assembly-ul curent;

protected internal - reuniunea precedentelor două;

private - accesabil doar în clasa conţinătoare; este specificatorul implicit.

3.9 Constructori de instanţă

Un constructor de instanţă este un membru care implementează acţiuni care sunt cerute pentru a iniţializa o instanţă a unei clase. Declararea unui astfel de constructor se face în felul următor:
atribute$_{opt}$ modificatori-de-constructor declarator-de-constructor corp-constructor
Un modificator de constructor poate fi: *public, protected, internal, private, extern*. Un declarator de constructor are forma:
nume-clasa (lista-parametrilor-formali$_{opt}$) initializator-de-constructor$_{opt}$
unde initializatorul-de-constructor are forma:
: base(lista-argumente$_{opt}$) sau
: this(lista-argumente$_{opt}$).
Corp-constructor poate fi: un bloc de declaraţii şi instrucţiuni delimitat de acolade sau caracterul punct şi virgulă.

Un constructor are acelaşi nume ca şi clasa din care face parte şi nu returnează un tip. Constructorii de instanţă nu se moştenesc. Dacă o clasă nu conţine nici o declaraţie de constructor de instanţă, atunci compilatorul va crea automat unul implicit. O clasă care este moştenită dintr-o altă clasă ce nu are constructori fără parametri va trebui să utilizeze un apel de constructor de clasă de bază pentru care să furnizeze parametrii potriviţi; acest apel se face prin intermediul iniţializatorului de constructor. Un constructor poate apela la un alt constructor al clasei din care face parte pentru a efectua iniţializări. Când există câmpuri instanţă care au o expresie de iniţializare în afara constructorilor clasei respective, atunci aceste iniţializări se vor face înainte de apelul de constructor al clasei de bază.

3.10 Câmpuri

Un câmp reprezintă un membru asociat cu un obiect sau cu o clasă. Modificatorii de câmp care se pot specifica opţional înaintea unui câmp sunt cei de mai sus, la care se adaugă modificatorii *new, readonly, volatile, static*,

ce vor fi prezentaţi mai jos. Pentru orice câmp este necesară precizarea unui tip de date, ce trebuie să aibe gradul de accesibilitate cel puţin cu al câmpului ce se declară. Opţional, câmpurile pot fi iniţializate cu valori compatibile. Un astfel de câmp se poate folosi fie prin specificarea numelui său, fie printr-o calificare bazată pe numele clasei sau al unui obiect.

Exemplu:

```
class A
{
  int a;//acces implicit de tip privat
  static void Main()
  {
    A objA = new A();
    objA.a = 1;//se poate accesa in interiorul clasei
  }
}
```

3.10.1 Câmpuri instanţe

Dacă o declaraţie de câmp nu include modificatorul static, atunci acel câmp se va regăsi în orice obiect de tipul clasei curente care va fi instanţiat. Modificări ale acestor câmpuri se vor face independent pentru fiecare obiect. Deoarece un astfel de câmp are o valoare specifică fiecărui obiect, accesarea lui se va face prin calificarea cu numele obiectului:

```
objA.a = 1;
```

(dacă modificatorii de acces permit aşa ceva). În interiorul unui instanţe de clase se poate folosi cuvântul *this*, reprezentând referinţă la obiectul curent.

3.10.2 Câmpuri statice

Când o declaraţie de câmp include un specificator static, câmpul respectiv nu apaţine fiecărei instanţe în particular, ci clasei însăşi. Accesarea unui câmp static din exteriorul clasei nu se face prin intermediul unui obiect, ci prin numele clasei:

```
class B
{
  public static int V = 3;
  static void Main()
  {
```

```
    B.V++;//corect
    V++;//corect
    B b = new B();
    b.V++//incorect
  }
}
```

Dacă se face calificarea unui astfel de câmp folosind un nume de obiect se semnalează o eroare de compilare.

3.10.3 Câmpuri readonly

Declararea unui câmp de tip *readonly* (static sau nu) se face prin specificarea cuvântului *readonly* în declaraţia sa:

```
class A
{
  public readonly string salut = ‘‘Salut’’;
  public readonly string nume;
  public class A(string nume)
  {
    this.nume = nume;
  }
}
```

Atribuirea asupra unui câmp de tip *readonly* se poate face doar la declararea sa (ca în exemplu de mai sus) sau prin intermediul unui constructor. Valoarea unor astfel de câmpuri nu e obligatorie a fi cunoscute la compilare.

3.10.4 Câmpuri volatile

Modificatorul "volatile" se poate specifica doar pentru tipurile:

- byte, sbyte, short, ushort, int, uint, char, float, bool;
- un tip enumerare având tipul de reprezentare byte, sbyte, short, ushort, int, uint;
- un tip referinţă

Pentru câmpuri nevolatile, tehnicile de optimizare care reordonează instrucţiunile pot duce la rezultate neaşteptate sau nepredictibile în programe multithreading care accesează câmpurile fără sincronizare (efectuabilă cu instrucţiunea lock).

Aceste optimizări pot fi făcute de către compilator, de către sistemul de rulare[5] sau de către hardware. Următoarele tipuri de optimizări sunt afectate în prezenţa unui modificator volatile:

- citirea unui câmp *volatile* este garantată că se va întâmpla înainte de orice referire la câmp care apare după citire;
- orice scriere a unui câmp *volatile* este garantată că se va petrece după orice instrucţiune anterioară care se referă la câmpul respectiv.

3.10.5 Iniţializarea câmpurilor

Pentru fiecare câmp declarat se va asigna o valoare implicită astfel:

- numeric: 0
- bool: false
- char: `'\0'`
- enum: 0
- referinţă: null

3.11 Constante

O constantă este un câmp a cărui valoare poate fi calculată la compilare. O constantă poate fi prefixată de următorii modificatori: *new, public, protected, internal, protected internal,private*. Cuvantul *new* poate să se combine cu unul din ceilalţi 4 modificatori de acces. Pentru un câmp constant e obligatoriu să se asigneze o valoare calculabilă la compilare:

```
class A
{
  public const int n=2;
}
```

Tipul unei constante poate fi *sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double decimal, bool, string, enum*, referinţă. Valoarea care se asigneză unei constante trebuie să admită o conversie implicită către tipul constantei. Tipul unei constante trebuie să fie cel puţin la fel de accesibil ca şi constanta însăşi.

[5]Engl: runtime system

Orice câmp constant este automat un câmp static. Un câmp constant diferă de un câmp static *readonly*: *const*-ul are o valoare cunoscută la compilare, pe când valoarea unui *readonly* poate fi iniţializată la runtime în interiorul constructorului (cel mai târziu, de altfel).

3.12 Metode

O metodă este un membru care implementează o acţiune care poate fi efectuată de către un obiect sau o clasă. Antetul unei metode se declară în felul următor:
atribute$_{opt}$ modificator-de-metoda$_{opt}$ tip-de-retur nume (lista-parametrilor-formali$_{opt}$) corp-metoda
unde modificator-de-metoda poate fi:

- orice modificator de acces
- new
- static
- virtual
- sealed
- override
- abstract
- extern

Tipul de retur poate fi orice tip de dată care este cel puţin la fel de accesibil ca şi metoda însăşi sau void (absenţa informaţiei returnate); nume poate fi un identificator de metodă din clasa curentă sau un identificator calificat cu numele unei interfeţe pe care o implementează (*NumeInterfata.numeMetoda*); parametrii pot fi de tip *ref, out, params*, sau fără nici un calificator; corp-metoda este un bloc cuprins între acolade sau doar caracterul ";" (dacă este vorba de o metodă ce nu se implementează în tipul curent).

Despre calificatorii *virtual, override, sealed, new, abstract* se va discuta mai pe larg într–o secţiune viitoare.

3.12.1 Metode statice şi nestatice

O metodă se declară a fi statică dacă numele ei este prefixat cu modificatorul de metodă static. O astfel de metodă nu operează asupra unei instanţe anume, ci doar asupra clasei. Este o eroare ca o metodă statică să facă referire la un membru nestatic al unei clase. Apelul unei astfel de metode se face prin *NumeClasa.NumeMetoda* sau direct *NumeMetoda* dacă este apelată din context static al aceleiaşi clase (de exemplu de către o metodă statică, dar se poate şi dintr–o clasă imbricată — a se vedea secţiunea dedicată 4.5).

O metodă nestatică nu are cuvântul "static" specificat; ea este apelabilă pentru un obiect anume.

3.12.2 Metode externe

Metodele externe se declară folosind modificatorul extern; acest tip de metode sunt implementate extern, de obicei în alt limbaj decât C#. Deoarece o astfel de metodă nu conţine o implementare, corpul acestei metode este ";".

Exemplu: se utilizează metoda MessageBox importată din biblioteca dll User32.dll:

```
using System;
using System.Runtime.InteropServices;
class Class1
{
  [DllImport("User32.dll")]
  public static extern int MessageBox(int h, string m, string c,
    int type);
  static void Main(string[] args)
  {
    int retVal = MessageBox(0, "Hello", "Caption", 0);
  }
}
```

Curs 4

Clase (continuare)

4.1 Proprietăţi

O proprietate este un membru care permite acces la partea de stare a unei clase. Exemple de proprietăţi sunt: lungimea unui şir, numele unui client, textul continut într–un control de tip TextBox. Proprietăţile sunt extensii naturale ale câmpurilor, cu deosebirea că ele nu presupun alocarea de memorie. Ele sunt de fapt nişte metode (accesori) care permit citirea sau setarea unor atribute ale unui obiect sau clase; reprezintă modalitatea de scriere a unor metode de tip get/set pentru clase sau obiecte.

Declararea unei proprietăţi se face astfel:
modificator-de-proprietate$_{opt}$ tip numeproprietate definitie-get$_{opt}$ definitie-set$_{opt}$
unde modificator-de-proprietate este: atribute$_{opt}$ modificator-de-acces$_{opt}$ get corp-get
atribute$_{opt}$ modificator-de-acces$_{opt}$ set corp-set

Modificatorii de acces sunt: *protected, internal, private, protected internal, public.*

Tipul unei proprietăţi specifică tipul de dată ce poate fi accesat, i.e. ce valori vor putea fi atribuite proprietăţii respective (dacă accesorul de tip *set* a fost definit), respectiv care este tipul valorii returnate de această proprietate (corespunzător accesorului de tip *get*).

Exemplu:

```
using System;
class Circle
{
  private double radius;
  public double Radius
  {
```

```
    get
    {
      return radius;
    }
    set
    {
      radius = value;
    }
  }
  public double Area
  {
    get
    {
      return Math.PI * radius * radius;
    }
    set
    {
      radius = Math.Sqrt(value/Math.PI);
    }
  }
}

class Test
{
  static void Main()
  {
    Circle c = new Circle();
    c.Radius = 10;
    Console.WriteLine(‘‘Area: {0}’’, c.Area);
    c.Area = 15;
    Console.WriteLine(‘‘Radius: {0}’’. c.Radius)
  }
}
```

Un accesor de tip *get* corespunde unei metode fără parametri, care returnează o valoare de tipul proprietăţii. Când o proprietate este folosită într–o expresie, accesorul *get* este o apelat pentru a returna valoarea cerută.

Un accesor de tip *set* corespunde unei metode cu un singur parametru de tipul proprietăţii şi tip de retur *void*. Acest parametru implicit al lui *set* este numit întotdeauna *value*. Când o proprietate este folosită ca destinatar într–o atribuire, sau când se folosesc operatorii ++ şi −−, accesorului *set* i

se transmite un parametru care reprezintă noua valoare.

În funcţie de prezenţa sau absenţa accesorilor, o proprietate este clasificată după cum urmează:

proprietate read–write, dacă are ambele tipuri de accesori;

proprietate read–only, dacă are doar accesor de tip *get*; este o eroare de compilare să se facă referire în program la o proprietate în sensul în care s–ar cere operarea cu un accesor de tip *set*;

proprietate write–only, dacă este prezent doar accesorul de tip *set*; este o eroare de compilare utilizarea unei proprietăţi într–un context în care ar fi necesară prezenţa accesorului *get*.

Există cazuri în care se doreşte ca un accesor sa aibă un anumit grad de acces (public, de exemplu), iar celalat alt tip de acces (e.g. protected). Începând cu .NET Framework 2.0, acest lucru este posibil:

```
public class Employee
{
  private string name;
  public Employee(string name)
  {
    this.name = name;
}
  public string Name
  {
    get { return name; }
    protected set { name = value; }
  }
}
```

Într–un asemenea caz, trebuie respectată următoarea regulă: întreaga proprietate trebuie să fie declarată cu grad de acces mai larg decât accesorul pentru care se restricţionează gradul de acces.

Demn de menţionat este că proprietăţile pot fi folosite nu doar pentru a asigura o sintaxă simplu de folosit pentru metodele tradiţionale de tip get/set, ci şi pentru scrierea controalelor .NET utilizator.

În figura 4.1 este dată reprezentarea unui control utilizator:

Codul corespunzător este dat mai jos:

```
using System;
using System.Collections;
```

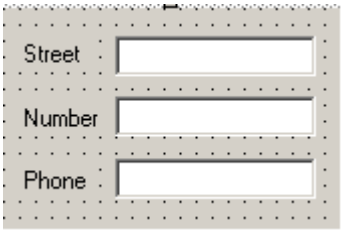

Figura 4.1: Control definit de utilizator

```
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Windows.Forms;

namespace UserControlSample
{
  public class UserControl1 : System.Windows.Forms.UserControl
  {
    private System.Windows.Forms.Label label1;
    private System.Windows.Forms.TextBox streetTextBox;
    private System.Windows.Forms.Label label2;
    private System.Windows.Forms.TextBox numberTextBox;
    private System.Windows.Forms.Label label3;
    private System.Windows.Forms.TextBox phoneTextBox;
    private System.ComponentModel.Container components = null;
    public UserControl1()
    {
      // This call is required by the Windows.Forms Form Designer.
      InitializeComponent();
    }

    protected override void Dispose( bool disposing )
    {
      if( disposing )
      {
         if( components != null )
           components.Dispose();
      }
      base.Dispose( disposing );
    }
```

```
#region Component Designer generated code

private void InitializeComponent()
{
  this.label1 = new System.Windows.Forms.Label();
  this.streetTextBox = new System.Windows.Forms.TextBox();
  this.label2 = new System.Windows.Forms.Label();
  this.numberTextBox = new System.Windows.Forms.TextBox();
  this.label3 = new System.Windows.Forms.Label();
  this.phoneTextBox = new System.Windows.Forms.TextBox();
  this.SuspendLayout();
  this.label1.AutoSize = true;
  this.label1.Location = new System.Drawing.Point(8, 16);
  this.label1.Name = "label1";
  this.label1.Size = new System.Drawing.Size(34, 13);
  this.label1.TabIndex = 0;
  this.label1.Text = "Street";
  this.streetTextBox.Location = new System.Drawing.Point(56, 14);
  this.streetTextBox.Name = "streetTextBox";
  this.streetTextBox.TabIndex = 1;
  this.streetTextBox.Text = "";
  this.label2.AutoSize = true;
  this.label2.Location = new System.Drawing.Point(8, 48);
  this.label2.Name = "label2";
  this.label2.Size = new System.Drawing.Size(44, 13);
  this.label2.TabIndex = 2;
  this.label2.Text = "Number";
  this.numberTextBox.Location = new System.Drawing.Point(56, 44);
  this.numberTextBox.Name = "numberTextBox";
  this.numberTextBox.TabIndex = 3;
  this.numberTextBox.Text = "";
  this.label3.AutoSize = true;
  this.label3.Location = new System.Drawing.Point(8, 79);
  this.label3.Name = "label3";
  this.label3.Size = new System.Drawing.Size(37, 13);
  this.label3.TabIndex = 4;
  this.label3.Text = "Phone";
  this.phoneTextBox.Location = new System.Drawing.Point(56, 75);
  this.phoneTextBox.Name = "phoneTextBox";
  this.phoneTextBox.TabIndex = 5;
  this.phoneTextBox.Text = "";
```

```
      this.Controls.AddRange(new System.Windows.Forms.Control[] {
      this.phoneTextBox,
      this.label3,
      this.numberTextBox,
      this.label2,
      this.streetTextBox,
      this.label1});
      this.Name = "UserControl1";
      this.Size = new System.Drawing.Size(168, 112);
      this.ResumeLayout(false);
    }
    #endregion

    [Category ("Data"), Description ("Contents of Street Control")]
    public string Street
    {
      get{ return streetTextBox.Text; }
      set{ streetTextBox.Text = value; }
    }

    [Category ("Data"), Description ("Contents of Number Control")]
    public string Number
    {
      get{ return numberTextBox.Text; }
      set{ numberTextBox.Text = value; }
    }

    [Category ("Data"), Description ("Contents of Phone Control")]
    public string Phone
    {
      get{ return phoneTextBox.Text; }
      set{ phoneTextBox.Text = value; }
    }
  }
}
```

Interesante sunt aici proprietăţile publice Street, Number şi Phone care vor fi vizibile în fereastra *Properties* atunci când acest control va fi adăugat la o formă. Atributele cuprinse între paranteze drepte sunt opţionale, dar vor face ca aceste proprietăţi să fie grupate în secţiunea de date a ferestrei *Properties*, şi nu în cea "Misc".

4.2 Indexatori

Uneori are sens tratarea unui obiect ca fiind un vector de elemente. Un indexator este o generalizare a supraîncărcării operatorului [] din C++.

Declararea unui indexator se face în felul următor:
atribute$_{opt}$ modificatori-de-indexator$_{opt}$ declarator-de-indexator {declaratii-de-accesori}

Modificatorii de indexator pot fi: *new, public, protected, internal, private, protected internal, virtual, sealed, override, abstract, extern.* Declaratorul de indexator are forma:
tip-de-retur this[lista-parametrilor-formali].
Lista parametrilor formali trebuie să conţină cel puţin un parametru şi nu poate să aibe vreun parametru de tip *ref* sau *out.* Declaraţiile de accesor vor conţine accesor get sau accesor set, asemănător cu cei de la proprietăţi.

Exemple:

1. Exemplul 1: un indexator simplu:

```
using System;
class MyVector
{
  private double[] v;
  public MyVector( int length )
  {
    v = new double[ length  ];
  }
  public int Length
  {
    get
    {
      return length;
    }
  }
  public double this[int index]
  {
    get
    {
      return v[ index];
    }
    set
    {
```

```
      v[index] = value;
    }
  }
}

class Test
{
  static void Main()
  {
    MyVector v = new MyVector( 10 );
    v[0] = 0;
    v[1] = 1;
    for( int i=2; i<v.Length; i++)
    {
      v[i] = v[i-1] + v[i-2];
    }
    for( int i=0; i<v.Length; i++)
    {
      Console.WriteLine(‘‘v[‘‘ + i.ToString() + ‘‘]=’’ + v[i]);
    }
  }
}
```

2. Exemplul 2: supraîncărcarea indexatorilor:

```
using System;
using System.Collections;
class DataValue
{
  public DataValue(string name, object data)
  {
    this.name = name;
    this.data = data;
  }
  public string Name
  {
    get
    {
      return(name);
    }
    set
```

```
    {
      name = value;
    }
  }

  public object Data
  {
    get
    {
      return(data);
    }
    set
    {
      data = value;
    }
  }
  string name;
  object data;
}

class DataRow
{
  ArrayList row;
  public DataRow()
  {
    row = new ArrayList();
  }

  public void Load()
  {
    row.Add(new DataValue("Id", 5551212));
    row.Add(new DataValue("Name", "Fred"));
    row.Add(new DataValue("Salary", 2355.23m));
  }

  public object this[int column]
  {
    get
    {
      return(row[column - 1]);
    }
```

```
    set
    {
      row[column - 1] = value;
    }
  }

  private int findColumn(string name)
  {
    for (int index = 0; index < row.Count; index++)
    {
      DataValue dataValue = (DataValue) row[index];
      if (dataValue.Name == name)
      return(index);
    }
    return(-1);
  }

  public object this[string name]
  {
    get
    {
      return this[findColumn(name)];
    }
    set
    {
      this[findColumn(name)] = value;
    }
  }
}

class Test
{
  public static void Main()
  {
    DataRow row = new DataRow();
    row.Load();
    DataValue val = (DataValue) row[0];
    Console.WriteLine("Column 0: {0}", val.Data);
    val.Data = 12; // set the ID
    DataValue val = (DataValue) row["Id"];
    Console.WriteLine("Id: {0}", val.Data);
```

```
    Console.WriteLine("Salary: {0}",
    ((DataValue) row["Salary"]).Data);
    ((DataValue)row["Name"]).Data = "Barney"; // set the name
    Console.WriteLine("Name: {0}", ((DataValue) row["Name"]).Data);
  }
}
```

3. Exemplul 3: Indexator cu mai mulţi parametri:

```
using System;
namespace MyMatrix
{
  class Matrix
  {
    double[,] matrix;

    public Matrix( int rows, int cols )
    {
      matrix = new double[ rows, cols];
    }

    public double this[int i, int j]
    {
      get
      {
        return matrix[i,j];
      }
      set
      {
        matrix[i,j] = value;
      }
    }

    public int RowsNo
    {
      get
      {
        return matrix.GetLength(0);
      }
    }
```

```
        public int ColsNo
        {
          get
          {
            return matrix.GetLength(1);
          }
        }

        static void Main(string[] args)
        {
          MyMatrix m = new MyMatrix(2, 3);
          Console.WriteLine("Lines: {0}", m.RowsNo);
          Console.WriteLine("Columns: {0}", m.ColsNo);
          for(int i=0; i<m.RowsNo; i++)
          {
            for( int j=0; j<m.ColsNo; j++)
            {
              m[i,j] = i + j;
            }
          }
          for(int i=0; i<c.RowsNo; i++)
          {
            for( int j=0; j<c.ColsNo; j++)
            {
              Console.Write(c[i,j] + " ");
            }
            Console.WriteLine();
          }
        }
      }
    }
```

Remarcăm ca accesarea elementelor se face prin perechi de tipul get/set, precum la proprietăţi. Ca şi în cazul proprietăţilor, este posibil ca un accesor să aibe un alt grad de acces decât celălalt, folosind acelasi mecanism: se declară indexatorul ca având un anumit grad de accesibilitate, iar pentru un accesor se va declara un grad de acces mai restrictiv.

4.3 Operatori

Un operator este un membru care defineşte semnificaţia unei expresii operator care poate fi aplicată unei instanţe a unei clase. Corespunde supraîncărcării operatorilor din C++. O declaraţie de operator are forma:
atribute$_{opt}$ modificatori-de-operator declaratie-de-operator corp-operator
Se pot declara operatori unari, binari şi de conversie.

Următoarele reguli trebuie să fie respectate pentru orice operator:

1. Orice operator trebuie să fie declarat public şi static.

2. Parametrii unui operator trebuie să fie transmişi prin valoare;

3. Acelaşi modificator nu poate apărea de mai multe ori în antetul unui operator

4.3.1 Operatori unari

Supraîncărcarea operatorilor unari are forma:
tip operator operator-unar-supraincarcabil (tip identificator) corp
Operatorii unari supraîncărcabili sunt: + - ! ~ ++ – true false. Următoarele reguli trebuie să fie respectate la supraîncărcarea unui operator unar (T reprezintă clasa care conţine definiţia operatorului):

1. Un operator +, -, !, ~ trebuie să preia un singur parametru de tip T şi poate returna orice tip.

2. Un operator ++ sau – trebuie să preia un singur parametru de tip T şi trebuie să returneze un rezultat de tip T.

3. Un operator unar true sau false trebuie să preia un singur parametru de tip T şi să returneze bool.

Operatorii true şi false trebuie să fie ori ambii definiţi, ori nici unul (altfel apare o eroare de compilare). Ei sunt necesari pentru cazuri de genul:

```
if( a==true )
```

sau

```
if( a==false )
```

Mai general, ei pot fi folosiţi ca expresii de control în if, do, while şi for, precum şi în operatorul ternar "? :". Deşi pare paradoxal, nu este obligatoriu ca `if (a==true)` să fie echivalentă cu `if (!(a==false))`, de exemplu pentru tipuri SQL care pot avea valoare de nul, ceea ce nu înseamă nici true, nici false, ci lipsă de informaţie.

Exemplu:

```
public struct DBBool
{
  private int x;
  public static bool operator true(DBBool x)
  {
    return x.value > 0;
  }
  public static bool operator false(DBBool x)
  {
    return x.value <= 0;
  }
  ...
}
```

Exemplul de mai jos arată modul în care se face supraîncărcarea operatorului ++, care poate fi folosit atât ca operator de preincrementare cât şi ca operator de postincrementare:

```
public class IntVector
{
  public int Length { ... } // read-only property
  public int this[int index] { ... } // read-write indexer
  public IntVector(int vectorLength) { ... }
  public static IntVector operator++(IntVector iv)
  {
    IntVector temp = new IntVector(iv.Length);
    for (int i = 0; i < iv.Length; ++i)
      temp[i] = iv[i] + 1;
    return temp;
  }
}

class Test
{
  static void Main()
```

```
  {
    IntVector iv1 = new IntVector(4); // vector of 4x0
    IntVector iv2;
    iv2 = iv1++; // iv2 contains 4x0, iv1 contains 4x1
    iv2 = ++iv1; // iv2 contains 4x2, iv1 contains 4x2
  }
}
```

4.3.2 Operatori binari

Declararea unui operator binar se face astfel:
tip operator operator-binar-supraincarcabil (tip identificator, tip identificator)
corp
Operatorii binari supraîncărcabili sunt: `+ - * / % & | ^ << >> == != > < >= <=`.
Cel puţin unul dintre cei doi parametri preluaţi trebuie să fie de tipul conţinător.
Operatorii de shiftare trebuie să aibă primul parametru de tipul clasei în care se declară, iar al doilea parametru de tip int. Unii operatori trebuie să se declare în pereche:

1. operatorii == şi `!=`
2. operatorii > şi <
3. operatorii >= şi <=

Pentru operaratorul ==, este indicată şi definirea metodei Equals(), deoarece tipul respectiv va putea fi astfel folosit şi de către limbaje care nu suportă supraîncărcarea operatorilor, dar pot apela metoda polimorfică Equals() definită în clasa object.

Nu se pot supaîncărca operatorii $+=, -=, /=, *=$; dar pentru ca aceştia să funcţioneze, este suficient să se supraîncarce operatorii corespunzători: $+, -, /, *$.

4.3.3 Operatori de conversie

O declaraţie de operator de conversie trebuie introduce o conversie definită de utilizator, care se va adăuga (dar nu va suprascrie) la conversiile predefinite.
Declararea unui operator de conversie se face astfel:
implicit operator tip (tip parametru) corp
explicit operator tip (tip parametru) corp
După cum se poate deduce, conversiile pot fi implicite sau explicite. Un astfel de operator va face conversia de la un tip sursă, indicat de tipul parametrului

din antet la un tip destinaţie, indicat de tipul de retur. O clasă poate să declare un operator de conversie de la un tip sursă S la un tip destinaţie T cu următoarele condiţii:

1. S şi T sunt tipuri diferite
2. Unul din cele două tipuri este clasa în care se face definirea.
3. T şi S nu sunt object sau tip interfaţă.
4. T şi S nu sunt baze una pentru cealaltă.

Un bun design asupra operatorilor de conversie are în vedere următoarele:

- Conversiile implicite nu ar trebui să ducă la pierdere de informaţie sau la apariţia de excepţii;
- Dacă prima condiţie nu este îndeplinită, atunci neapărat trebuie declarată ca o conversie explicită.

Exemplu:

```
using System;
public class Digit
{
  byte value;
  public Digit(byte value)
  {
    if (value < 0 || value > 9) throw new ArgumentException();
    this.value = value;
  }
  public static implicit operator byte(Digit d)
  {
    return d.value;
  }
  public static explicit operator Digit(byte b)
  {
    return new Digit(b);
  }
}
```

Prima conversie este implicită pentru că nu va duce la pierderea de informaţie. Cea de doua poate să arunce o excepţie (via constructor) şi de aceea este declarată ca şi conversie explicită.

4.3.4 Exemplu: clasa Fraction

```
using System;
public class Fraction
{
  public Fraction(int numerator, int denominator)
  {
    Console.WriteLine("In constructor Fraction(int, int)");
    this.numerator=numerator;
    this.denominator=denominator;
  }
  public Fraction(int wholeNumber)
  {
    Console.WriteLine("In Constructor Fraction(int)");
    numerator = wholeNumber;
    denominator = 1;
  }

  public static implicit operator Fraction(int theInt)
  {
    System.Console.WriteLine("In conversie implicita la Fraction");
    return new Fraction(theInt);
  }
  public static explicit operator int(Fraction theFraction)
  {
    System.Console.WriteLine("In conversie explicita la int");
    return theFraction.numerator /
    theFraction.denominator;
  }
  public static bool operator==(Fraction lhs, Fraction rhs)
  {
    Console.WriteLine("In operator ==");
    if (lhs.denominator * rhs.numerator ==
      rhs.denominator * lhs.numerator )
    {
      return true;
    }
    return false;
  }
  public static bool operator !=(Fraction lhs, Fraction rhs)
  {
```

```
    Console.WriteLine("In operator !=");
    return !(lhs==rhs);
  }
  public override bool Equals(object o)
  {
    Console.WriteLine("In metoda Equals");
    if (! (o is Fraction) )
    {
      return false;
    }
    return this == (Fraction) o;
  }
  public static Fraction operator+(Fraction lhs, Fraction rhs)
  {
    Console.WriteLine("In operator+");
    // 1/2 + 3/4 == (1*4) + (3*2) / (2*4) == 10/8
    int firstProduct = lhs.numerator * rhs.denominator;
    int secondProduct = rhs.numerator * lhs.denominator;
    return new Fraction(
    firstProduct + secondProduct,
    lhs.denominator * rhs.denominator
    );
    //ar mai trebui facuta reducerea termenilor
  }
  public override string ToString( )
  {
    String s = numerator.ToString( ) + "/" +
    denominator.ToString( );
    return s;
  }
  private int numerator;
  private int denominator;
}
public class Tester
{
  static void Main( )
  {
    Fraction f1 = new Fraction(3,4);
    Console.WriteLine("f1: {0}", f1.ToString( ));
    Fraction f2 = new Fraction(2,4);
    Console.WriteLine("f2: {0}", f2.ToString( ));
```

```
    Fraction f3 = f1 + f2;
    Console.WriteLine("f1 + f2 = f3: {0}", f3.ToString( ));
    Fraction f4 = f3 + 5;
    Console.WriteLine("f3 + 5 = f4: {0}", f4.ToString( ));
    Fraction f5 = new Fraction(2,4);
    if (f5 == f2)
    {
      Console.WriteLine("F5: {0} == F2: {1}",
      f5.ToString( ),
      f2.ToString( ));
    }
  }
}
```

4.4 Constructor static

Un constructor static este un membru care implementează acţiunile cerute pentru iniţializara unei clase. Declararea unui constructor static se face ca mai jos:
atribute$_{opt}$ modificator-de-constructor-static identificator( ) corp
Modificatorii de constructori statici se pot da sub forma:
extern$_{opt}$ static sau
static extern$_{opt}$
Constructorii statici nu se moştenesc, nu se pot apela direct şi nu se pot supraîncărca. Un constructor static se va executa cel mult o dată într-o aplicaţie. Se garantează faptul că acest constructor se va apela înaintea primei creări a unei instanţe a clasei respective sau înaintea primului acces la un membru static. Acest apel este nedeterminist, necunoscându–se exact când sau dacă se va apela. Un astfel de constuctor nu are specificator de acces şi poate să acceseze doar membri statici.

Exemplu:

```
class Color
{
  public Color(byte red, byte green, byte blue)
  {
    this.red = red;
    this.green = green;
    this.blue = blue;
  }
  byte red;
```

```
  byte green;
  byte blue;

  public static readonly Color Red;
  public static readonly Color Green;
  public static readonly Color Blue;
  // constructor static
  static Color()
  {
    Red = new Color(255, 0, 0);
    Green = new Color(0, 255, 0);
    Blue = new Color(0, 0, 255);
  }
}
class Test
{
  static void Main()
  {
    Color background = Color.Red;
  }
}
```

4.5 Clase imbricate

O clasă conţine membri, iar în particular aceştia pot fi şi clase.
Exemplul 1:

```
using System;
class A
{
  class B
  {
    public static void F()
    {
      Console.WriteLine(‘‘A.B.F’’);
    }
  }
  static void Main()
  {
    A.B.F();
  }
```

```
}
```

Exemplul 2:

```
public class List
{
  // Private data structure
  private class Node
  {
    public object Data;
    public Node Next;
    public Node(object data, Node next)
    {
      this.Data = data;
      this.Next = next;
    }
  }
  private Node first = null;
  private Node last = null;
  //Interfata publica
  public void AddToFront(object o) {...}
  public void AddToBack(object o) {...}
  public object RemoveFromFront() {...}
  public object RemoveFromBack() {...}
  public int Count { get {...} }
}
```

Accesarea unei clase imbricate se face prin *NumeClasaExterioara.NumeClasaInterioara* (aşa cum este arătat în Exemplul 1), de unde se deduce că o clasă imbricată se comportă ca un membru static al tipului conţinător. O clasă declarată în interiorul unei alte clase poate avea unul din gradele de accesibilitate *public, protected internal, protected, internal, private* (implicit este *private*). O clasă declarată în interiorul unei structuri poate fi declarată *public, internal* sau *private* (implicit *private*).

Crearea unei instanţe a unei clase imbricate nu trebuie să fie precedată de crearea unei instanţe a clasei exterioare conţinătoare, aşa cum se vede din Exemplul 1. O clasă imbricată nu are vreo relaţie specială cu membrul predefinit *this* al clasei conţinătoare. Altfel spus, nu se poate folosi *this* în interiorul unei clase imbricate pentru a accesa membri instanţă din tipul conţinător. Dacă o clasă imbricată are nevoie să acceseze membri instanţă ai clasei conţinătoare, va trebui să primească prin constructor parametrul *this* care să se referă la o astfel de instanţă:

```
using System;
class C
{
  int i = 123;
  public void F()
  {
    Nested n = new Nested(this);
    n.G();
  }

  public class Nested
  {
    C this_c;
    public Nested(C c)
    {
      this_c = c;
    }
    public void G()
    {
      Console.WriteLine(this_c.i);
    }
  }
}
class Test
{
  static void Main()
  {
    C c = new C();
    c.F();
  }
}
```

Se observă de mai sus că o clasă imbricată poate manipula toţi membrii din interiorul clasei conţinătoare, indiferent de gradul lor de accesibilitate. În cazul în care clasa exterioară (conţinătoare) are membri statici, aceştia pot fi utilizaţi fără a se folosi numele clasei conţinătoare:

```
using System;
class C
{
  private static void F()
  {
```

```
    Console.WriteLine("C.F");
  }
  public class Nested
  {
    public static void G()
    {
      F();
    }
  }
}
class Test
{
  static void Main()
  {
    C.Nested.G();
  }
}
```

Clasele imbricate se folosesc intens în cadrul containerilor pentru care trebuie să se construiască un enumerator. Clasa imbricată va fi în acest caz strâns legată de container şi va duce la o implementare uşor de urmărit şi de întreţinut.

4.6 Destructori

Managementul memoriei este făcut sub platforma .NET în mod automat, de către garbage collector, parte componentă a CLR–ului.

Acest mecanism de garbage collection scuteşte programatorul de grija dealocării memoriei. Dar există situaţii în care se doreşte să se facă management manual al dealocării resurselor (de exemplu al resurselor care ţin de sistemul de operare sau de servere: fişiere, conexiuni la reţea sau la serverul de baze de date, ferestre, etc, sau al altor resurse al căror management nu se face de către CLR). În C# există posibilitatea de a lucra cu destructori sau cu metode de tipul *Dispose(), Close()*.

Un destructor se declară în felul următor:
atribute_{opt} extern_{opt} ~identificator() corp-destructor
unde identificator este numele clasei. Un destructor nu are modificator de acces, nu poate fi apelat manual, nu poate fi supraîncărcat, nu este moştenit.

Un destructor este o scurtătură sintactică pentru metoda *Finalize()*, care este definită în clasa *System.Object*. Programatorul nu poate să suprascrie sau să apeleze această metodă.

Exemplu:

```
~MyClass()
{
   // Perform some cleanup operations here.
}
```

Metoda de mai sus este automat translatată în:

```
protected override void Finalize()
{
   try
   {
      // Perform some cleanup operations here.
   }
   finally
   {
      base.Finalize();
   }
}
```

Problema cu destructorul este că el e chemat doar de către garbage collector, dar acest lucru se face nedeterminist (cu toate că apelarea de destructor se face în cele din urmă, dacă programatorul nu împiedică explicit acest lucru).

Există cazuri în care programatorul doreşte să facă dealocarea manual, astfel încât să nu aştepte ca garbage collectorul să apeleze destructorul. Programatorul poate scrie o metoda care să facă acest lucru. Se sugerează definirea unei metode *Dispose()* care ar trebui să fie explicit apelată atunci când resurse de sistem de operare trebuie să fie eliberate. În plus, clasa respectivă ar trebui să implementeze interfaţa *System.IDisposable*, care conţine această metodă.

În acest caz, *Dispose()* ar trebui să inhibe executarea ulterioară a destructorului (care am vazut ca e de fapt un finalizator, metoda Finalize) pentru instanţa curentă. Această manevră permite evitarea eliberării unei resurse de două ori. Dacă clientul nu apelează explicit *Dispose()*, atunci garbage collectorul va apela el destructorul la un moment dat. Întrucât utilizatorul poate să nu apeleze *Dispose()*, este necesar ca tipurile care implemetează această metodă să definească de asemenea destructor.

Exemplu:

```
public class ResourceUser: IDisposable
{
```

```
  public void Dispose()
  {
    hwnd.Release();//elibereaza o fereastra in Win32
    GC.SuppressFinalization(this);//elimina apel de Finalize()
  }

  ~ResourceUser()
  {
    hwnd.Release();
  }
}
```

Pentru anumite clase C# se pune la dispoziţie o metodă numită *Close()* în locul uneia *Dispose()*: fisiere, socket-uri, ferestre de dialog, etc. Este indicat ca să se adauge o metodă *Close()* care să facă doar apel de *Dispose()*:

```
//in interiorul unei clase
public void Close()
{
  Dispose();
}
```

Modalitatea cea mai indicată este folosirea unui bloc *using*, caz în care se va elibera obiectul alocat (via metoda Dispose()) la sfârşitul blocului:

```
using( obiect )
{
  //cod
}//aici se va apela automat metoda Dispose()
```

4.7 Clase statice

Începând cu versiunea 2.0 a platformei .NET s-a introdus posibilitatea de a defini clase statice. Acest tip de clasă se foloseşte atunci când se doreşte accesarea membrilor fără a fi nevoie să se lucreze cu obiecte; se pot folosi acolo unde buna funcţionare nu este dependentă de starea unor instanţe.

Pentru a crea o clasă statică se foloseşte cuvântul `static` în declaraţia de clasă:

```
static class MyStaticClass
{
 //membri statici
}
```

Orice clasă statică are următoarele proprietăţi:

1. nu poate fi instanţiată
2. nu poate fi moştenită (este automat *sealed*, vezi secţiunea 4.9)
3. conţine doar membri statici

Exemplu:

```
public static class TemperatureConverter
{
 public static double CelsiusToFahrenheit(string temperatureCelsius)
 {
  double celsius = Double.Parse(temperatureCelsius);
  double fahrenheit = (celsius * 9 / 5) + 32;
  return fahrenheit;
 }
 public static double FahrenheitToCelsius(string temperatureFahrenheit)
 {
  double fahrenheit = Double.Parse(temperatureFahrenheit);
  //Convert Fahrenheit to Celsius.
  double celsius = (fahrenheit - 32) * 5 / 9;
  return celsius;
 }
}
class TestTemperatureConverter
{
 static void Main()
 {
  Console.WriteLine("Optiuni");
  Console.WriteLine("1. Celsius->Fahrenheit.");
  Console.WriteLine("2. Fahrenheit->Celsius.");
  Console.Write(":");
  string selection = Console.ReadLine();
  double f, c = 0;
  switch (selection)
  {
   case "1":
    Console.Write("Temperatura Celsius: ");
    f = TemperatureConverter.CelsiusToFahrenheit(Console.ReadLine());
    Console.WriteLine("Temperatura in Fahrenheit: {0:F2}", f);
    break;
```

```
   case "2":
    Console.Write("Temperatura Fahrenheit: ");
    c = TemperatureConverter.FahrenheitToCelsius(Console.ReadLine());
    Console.WriteLine("Temperature in Celsius: {0:F2}", c);
   break;
  }
 }
}
```

4.8 Specializarea şi generalizarea

Specializarea reprezintă o tehnică de a obţine noi clase pornind de la cele existente. Deseori între clasele pe care le modelăm putem observa relaţii de genul "este un/o": un om este un mamifer, un salariat este un angajat, etc. Toate acestea duc la crearea unei ierarhii de clase, în care din clase de bază (mamifer sau angajat) descind alte clase, care pe lângă comportament din clasa de bază mai au şi caracteristici proprii. Obţinerea unei clase derivate plecând de la altă clasă se numeşte specializare iar operaţia inversă se numeşte generalizare. O clasă de bază defineşte un tip comun, compatibil cu oricare din clasele derivate (direct sau indirect).

În C# o clasă nu trebuie să moştenească explicit din altă clasă; în acest caz se va considera că ea este implicit derivată din clasa predefinită *object* (tot una cu *Object*). C# nu permite moştenire multiplă, eliminând astfel complicaţiile întâlnite în C++. Ca alternativă, se permite totuşi implementarea de mai multe interfeţe.

4.8.1 Specificarea moştenirii

În C# se pentru o clasă D se defineşte clasa de bază B folosind următoarea formulă:

```
class D: B
{
 //declaratii si instructiuni
}
```

Dacă pentru o anumită clasă nu se specifică două puncte urmate de numele unei clase de bază atunci *object* va deveni bază pentru clasa în cauză.

Exemplu:

```
//clasa de baza in C#
```

```
public class Employee
{
 protected string name;
 protected string ssn;
}

//clasa derivata in C#
public class Salaried : Employee
{
 protected double salary;
 public Salaried( string name, string ssn, double salary )
 {
  this.name = name;
  this.ssn = ssn;
  this.salary = salary;
 }
}
```

Se observă că câmpurile *name* şi *ssn* din clasa de bază sunt accesibile în clasa derivată, datorită specificatorului de acces *protected*.

4.8.2 Apelul constructorilor din clasa de bază

În exemplul anterior nu s–a definit nici un constructor în clasa de bază *Employee*; constructorul clasei derivate trebuie să facă iniţializările câmpurilor în conformitate cu parametrii transmişi, chiar dacă o parte din aceste câmpuri provin din clasa de bază. Mai logic ar fi ca în clasa de bază să se găsească un constructor care să iniţializeze câmpurile proprii: *name* şi *ssn*. Întrucât constructorii nu se moştenesc, e nevoie ca în clasa derivată să se facă un apel explicit al constructorului clasei de bază. Acest apel se face prin *iniţializator de constructor* care are forma: două puncte urmate de *base(parametrii-efectivi)*.

```
public class Employee
{
 protected string name;
 protected string ssn;
 public Employee( string name, string ssn)
 {
  this.name = name;
  this.ssn = ssn;
  System.Console.WriteLine(‘‘Employee constructor: {0}, {1}’’,
```

```
  name, ssn);
 }
}
public class Salaried : Employee
{
 protected double salary;
 public Salaried(string name, string ssn, double salary):
  base(name, ssn)
 {
  this.salary = salary;
  System.Console.WriteLine(‘‘Salaried constructor: {0}’’,
   salary);
 }
}
class Test
{
 Salaried s = new Salaried(‘‘Jesse’’, ‘‘1234567890’’,
  100000.00);
}
```

La rulare se va obţine:

```
Employee constructor: Jesse, 1234567890
Salaried constructor: 100000.00
```

de unde se deduce că apelul de constructor de clasă de bază se face înaintea executării oricăror alte instrucţiuni conţinute în constructorul clasei derivate.

Dacă o clasă de bază nu are definit nici un constructor, atunci se va crea unul implicit (fără parametri). Dacă după un constructor al unei clase derivate nu se specifică un iniţializator de constructor, atunci va fi apelat constructorul implicit (fie creat automat de compilator, fie scris de către programator); dacă nu există nici un constructor implicit în clasa de bază, atunci programatorul trebuie să specifice un constructor din clasa de bază care va fi apelat, împreună cu parametrii adecvaţi.

4.8.3 Operatorii *is* şi *as*

Operatorul *is*

Operatorl *is* este folosit pentru a verifica dacă un anumit obiect este de un anumit tip. Este folosit de obicei înainte operaţiilor de downcasting (conversie explicită de la un tip de bază la unul derivat). Operatorul se foloseşte astfel:

```
instanta is NumeClasa
```

rezultatul acestei operaţii fiind true sau false.

Exemplu:

```
Employee e = ...;
if (e is Salaried)
{
 Salaried s = (Salaried)e;
}
```

În cazul în care s–ar face conversia explicită iar obiectul nu este de tipul la care se face conversia ar rezulta o excepţie: `System.InvalidCastException`.

Operatorul *as*

Acest operator este folosit pentru conversii explicite, returnând un obiect de tipul la care se face conversia sau null dacă conversia nu se poate face (nu se aruncă excepţii). Determinarea validităţii conversiei se face testând valoarea rezultată faţă de null: dacă rezultatul e null atunci conversia nu s–a putut face. Ca şi precedentul operator se foloseşte în special la downcasting.

Exemplu:

```
Employee e = ...;
Salaried s = e as Salaried;
if (s != null)
{
//se lucreaza cu instanta valida de tip Salaried
}
```

4.9 Clase *sealed*

Specificatorul *sealed* care se poate folosi înaintea cuvântului cheie *class* specifică faptul că clasa curentă nu se poate deriva. Este o eroare de compilare ca o clasă *sealed* să fie declarată drept clasă de bază.

Curs 5

Clase - polimorfism, clase abstracte. Structuri, interfeţe, delegaţi

5.1 Polimorfismul

Polimorfismul este capacitatea unei entităţi de a lua mai multe forme. În limbajul C# polimorfismul este de 3 feluri: parametric, ad–hoc şi de moştenire.

5.1.1 Polimorfismul parametric

Este cea mai slabă formă de polimorfism, fiind regăsită în majoritatea limbajelor. Prin polimorfismul parametric se permite ca o implementare de funcţie să poată prelucra orice număr de parametri. Acest lucru se poate obţine prin folosirea în C# a unui parametru de tip *params* (vezi 3.2).

5.1.2 Polimorfismul ad–hoc

Se mai numeşte şi supraîncărcarea metodelor, mecanism prin care în cadrul unei clase se pot scrie mai multe metode, având acelaşi nume, dar tipuri şi numere diferite de parametri de apel. Alegerea funcţiei care va fi apelată se va face la compilare, pe baza corespondenţei între tipurile parametrilor de apel şi tipurile parametrilor formali.

5.1.3 Polimorfismul de moştenire

Este forma cea mai evoluată de polimorfism. Dacă precedentele forme de polimorfism sunt aplicabile fără a se pune problema de moştenire, în acest caz este necesar să existe o ierarhie de clase. Mecanismul se bazează pe faptul că o clasă de bază defineşte un tip care este compatibil din punct de vedere al atribuirii cu orice tip derivat, ca mai jos:

```
class B{...}
class D: B{...}
class Test
{
 static void Main()
 {
  B b = new D();//upcasting=conversie implicita catre baza B
 }
}
```

Într–un astfel de caz se pune problema: ce se întâmplă cu metodele având aceeaşi listă de parametri formali şi care se regăsesc în cele două clase?

Să considerăm exemplul următor: avem o clasă *Shape* care conţine o metodă *public void Draw()*; din *Shape* se derivează clasa *Polygon* care implementează aceeaşi metodă în mod specific. Problema care se pune este cum se rezolvă un apel al metodei *Draw* în context de upcasting:

```
class Shape
{
 public void Draw()
 {
  System.Console.WriteLine(‘‘Shape.Draw()’’);
 }
}
class Polygon: Shape
{
 public void Draw()
 {
  System.Console.WriteLine(‘‘Polygon.Draw()’’);
  //desenarea s-ar face prin GDI+
 }
}
class Test
{
```

```
 static void Main()
 {
  Polygon p = new Polygon();
  Shape s = p;//upcasting
  p.Draw();
  s.Draw();
 }
}
```

La compilarea acestui cod se va obţine un avertisment:

```
warning CS0108: The keyword new is required on Polygon.Draw()
because it hides inherited member Shape.Draw()
```

dar despre specificatorul *new* vom vorbi mai jos (oricum, adăugarea lui nu va schimba cu nimic comportamentul de mai jos, doar va duce la dispariţia de avertisment). Codul de mai sus va afişa:

```
Polygon.Draw()
Shape.Draw()
```

Dacă prima linie afişată este conformă cu intuiţia, cea de-a doua pare la prima vedere ciudată, dar de fapt este perfect justificabilă: apelul de metodă *Draw()* este rezolvat în fiecare caz la compilare pe baza tipului declarat al obiectelor; ca atare apelul precedent este legat de corpul metodei *Draw* din clasa *Shape*, chiar dacă *s* a fost instanţiat de fapt pe baza unui obiect de tip *Polygon*.

Este posibil ca să se dorească schimbarea acestui comportament: apelul de metodă *Draw* să fie rezolvat în funcţie de tipul efectiv al obiectului care face acest apel, şi nu de tipul formal declarat. În cazul precedent, apelul *s.Draw()* trebuie să se rezolve de fapt ca fiind către metoda *Draw()* din *Polygon*, pentru că acesta este tipul la rulare al obiectului *s*. Cu alte cuvinte, apelul ar trebui să fie rezolvat la rulare şi nu la compilare, în funcţie de natura efectivă a obiectelor. Acest comportament polimorfic este referit sub denumirea *polimorfism de moştenire*.

5.1.4 *Virtual* şi *override*

Pentru a asigura faptul că legarea apelului de metode se face la rulare şi nu la compilare, e necesar ca în clasa de bază să se specifice că metoda *Draw()* este virtuală, iar în clasa derivată pentru aceeaşi metodă trebuie să se spună că este o suprascriere a celei din bază:

```
class Shape{
 public virtual void Draw(){...}
}
class Polygon{
 public override void Draw(){...}
}
```

În urma executării metodei *Main* din clasa de mai sus, se va afişa:

```
Polygon.Draw()
Polygon.Draw()
```

adică s–a apelat metoda corespunzătoare tipului efectiv de la rulare, în fiecare caz.

În cazul în care clasa *Polygon* este la rândul ei moştenită şi se doreşte ca polimorfismul să funcţioneze în continuare va trebui ca în această a treia clasă să suprascrie (*override*) metoda *Draw()*.

Un astfel de comportament polimorfic este benefic atunci când se foloseşte o colecţie de obiecte de tipul unei clase de bază:

```
Shape[] painting = new Shape[10];
painting[0] = new Shape();
painting[1] = new Polygon();
...
foreach( Shape s in painting)
 s.Draw();
```

5.1.5 Modificatorul *new* pentru metode

Modificatorul *new* se foloşeste pentru a indica faptul că o metodă dintr-o clasă derivată care are aceeaşi semnătură cu una dintr–o clasă de bază nu este o suprascriere polimorfică a ei, ci apare ca o nouă metodă. Este ca şi cum metoda declarată *new* ar avea nume diferit.

Să presupunem următorul scenariu: compania A crează o clasă *A* care are forma:

```
public class A{
 public void M(){
  Console.WriteLine(‘‘A.M()’’);
 }
}
```

O altă companie B va crea o clasă *B* care moşteneşte clasa A. Compania B nu are nici o influenţă asupra companiei A sau asupra modului în care aceasta va face modificări asupra clasei *A*. Ea va defini în interiorul clasei B o metodă *M()* şi una *N()*:

```
class B: A{
 public void M(){
  Console.WriteLine(‘‘B.M()’’);
  N();
  base.M();
 }
 protected virtual void N(){
  Console.WriteLine(‘‘B.N()’’);
 }
}
```

Atunci când compania B compilează codul, compilatorul C# va produce următorul avertisment:

```
warning CS0108: The keyword new is required on ‘B.M()’ because
it hides inherited member ‘A.M()’
```

Acest avertisment va notifica programatorul că clasa *B* defineşte o metodă *M()*, care va ascunde metoda *M()* din clasa de bază *A*. Această nouă metodă ar putea schimba înţelesul (semantica) lui *M()*, aşa cum a fost creat iniţial de compania A. Este de dorit în astfel de cazuri compilatorul să avertizeze despre posibile nepotriviri semantice. Oricum, programatorii din B vor trebui să pună în orice caz specificatorul *new* înaintea metodei *B.M()* pentru a elimina avertismentul.

Să presupunem că o aplicaţie foloseşte clasa *B()* în felul următor:

```
class App{
 static void Main(){
  B b = new B();
  b.M();
 }
}
```

La rulare se va afişa:

```
B.M()
B.N()
A.M()
```

Să presupunem că A decide adăugarea unei metode virtuale *N()* în clasa sa, metodă ce va fi apelată din *M()*:

```
public class A
{
 public void M()
 {
  Console.WriteLine(‘‘A.M()’’);
  N();
 }
 protected virtual void N()
 {
  Console.WriteLine(‘‘A.N()’’);
 }
}
```

La o recompilare făcută de B, este dat următorul avertisment:

```
warning CS0114: ‘B.N()’ hides inherited member ‘A.N()’. To make
the current member override that implementation, add the
override keyword. Otherwise, add the new keyword.
```

În acest mod compilatorul avertizează că ambele clase oferă o metodă *N()* a căror semantică poate să difere. Dacă B decide că metodele *N()* nu sunt semantic legate în cele două clase, atunci va specifica *new*, informând compilatorul de faptul că versiunea sa este una nouă, care nu suprascrie polimorfic metoda din clasa de bază.

Atunci când codul din clasa *App* este rulat, se va afişa la ieşire:

```
B.M()
B.N()
A.M()
A.N()
```

Ultima linie afişată se explică tocmai prin faptul că metoda *N()* din *B* este declarată *new* şi nu *override* (dacă ar fi fost override, ultima linie ar fi fost *B.N()*, din cauza polimorfismului).

Se poate ca B să decidă că metodele *M()* şi *N()* din cele două clase sunt legate semantic. În acest caz, ea poate şterge definiţia metodei *B.M*, iar pentru a semnala faptul că metoda *B.N()* suprascrie metoda omonimă din clasa părinte, va înlocui cuvântul *new* cu *override*. În acest caz, metoda *App.Main* va produce:

```
A.M()
B.N()
```

ultima linie fiind explicată de faptul că *B.N()* suprascrie o metodă virtuală.

5.1.6 Metode *sealed*

O metodă de tip *override* poate fi declarată ca fiind de tip *sealed*, astfel împiedicându–se suprascrierea ei într–o clasă derivată din cea curentă:

```
using System;
class A
{
 public virtual void F()
 {
  Console.WriteLine(‘‘A.F()’’);
 }
 public virtual void G()
 {
  Console.WriteLine(‘‘A.G()’’);
 }
}
class B: A
{
 sealed override public void F()
 {
  Console.WriteLine(‘‘B.F()’’);
 }
 override public void G()
 {
  Console.WriteLine(‘‘B.G()’’);
 }
}
class C: B
{
 override public void G()
 {
  Console.WriteLine(‘‘C.G()’’);
 }
}
```

Modificatorul *sealed* pentru *B.F* va împiedica tipul *C* să suprascrie metoda *F*.

5.1.7 Exemplu folosind *virtual, new, override, sealed*

Să presupunem următoare ierarhie de clase, reprezentată în Fig. 5.1; o clasă X moşteneste o clasă Y dacă sensul săgeţii este de la X la Y. Fiecare clasă are o metodă *void foo()* care determină afişarea clasei în care este definită şi pentru care se vor specifica *new, virtual, override, sealed.* Să

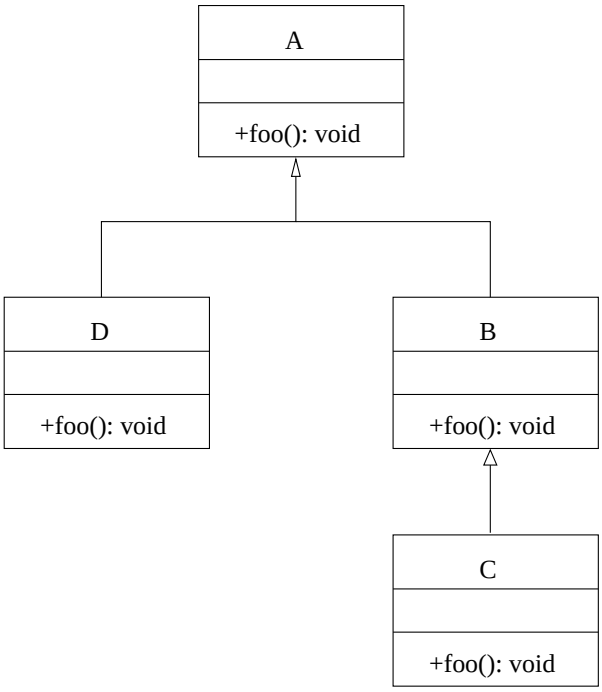

Figura 5.1: Ierarhie de clase

presupunem că clasa de test arată astfel:

```
public class Test
{
 static void Main()
 {
  A[] x = new A[4];
  x[0] = new A();
  x[1] = new B();
  x[2] = new C();
  x[3] = new D();

  A a = new A();
  B b = new B();
```

```
  C c = new C();
  D d = new D();

  /* secventa 1 */
  for(int i=0; i<4; i++)
  {
   x[i].foo();
  }
  /* secventa 2 */
  a.foo();
  b.foo();
  c.foo();
  d.foo();
 }
}
```

În funcţie de specificatorii metodelor *f()* din fiecare clasă, se obţin ieşirile din tabelul 5.1:

Tabelul 5.1: Efecte ale diferiţilor specificatori.

Metoda	A.f()	B.f()	C.f()	D.f()
Specificator	virtual	override	override	override
Ieşire secv. 1 Ieşire secv. 2	A.f A.f	B.f B.f	C.f C.f	D.f D.f
Specificator	virtual	override	new	override
Ieşire secv. 1 Ieşire secv. 2	A.f A.f	B.f B.f	B.f C.f	D.f D.f
Specificator	virtual	new	new	new
Ieşire secv. 1 Ieşire secv. 2	A.f A.f	A.f B.f	A.f C.f	A.f D.f
Specificator	virtual	new	override	override
Eroare de compilare deoarece C.f nu poate suprascrie metoda nevirtuală B.f()				
Specificator	virtual	virtual	override	override
Ieşire secv. 1 Ieşire secv. 2 Avertisment la compilare deoarece	A.f A.f	A.f B.f	A.f C.f	D.f D.f

Tabelul 5.1 (continuare)

Metoda	A.f()	B.f()	C.f()	D.f()
B.f înlocuieşte A.f				
Specificator	virtual	sealed override	override	override
Eroare de compilare deoarece deoarece B.f nu poate fi suprascrisă de C.f				

5.2 Clase şi metode abstracte

Deseori pentru o anumită clasă nu are sens crearea de instanţe, din cauza unei generalităţi prea mari a tipului respectiv. Spunem că această clasă este *abstractă*, iar pentru a împiedica efectiv crearea de instanţe de acest tip, se va specifica cuvântul *abstract* înaintea metodei. În exemplele de anterioare, clasele *Employee* şi *Shape* ar putea fi gândite ca fiind abstracte: ele conţin prea puţină informaţie pentru a putea crea instanţe utile.

Analog, pentru o anumită metodă din interiorul unei clase uneori nu se poate specifica o implementare. De exemplu, pentru clasa *Shape* de mai sus, este imposibil să se dea o implementare la metoda *Draw()*, tocmai din cauza generalităţii acestei clase. Ar fi util dacă pentru această metodă programatorul ar fi obligat să dea implementări specifice ale acestei metode pentru diversele clase derivate. Pentru a se asigura tratarea polimorfică a acestui tip abstract, orice metodă abstractă este automat şi virtuală. Orice metodă care este declarată abstractă implică declararea clasei ca fiind abstractă.

Exemplu:

```
abstract class Shape
{
 public abstract void Draw();
 //remarcam lipsa implementarii si semnul punct si virgula
}
```

Orice clasă care este derivată dintr–o clasă abstractă va trebui fie să nu aibă nici o metodă abstractă moştenită fără implementare, fie să se declare ca fiind abstractă. Existenţa de metode neimplementate nu va permite instanţierea clasei respective.

5.3 Tipuri parţiale

Începând cu versiunea 2.0 a platfomei .NET este posibil ca definiţia unei clase, interfeţe sau structuri să fie făcută în mai multe fişiere sursă. Definiţia clasei se obţine din reuniunea părţilor componente, lucru făcut automat de către compilator. Această spargere în fragmente este benefică în următoarele cazuri:

- atunci când se lucrează cu proiecte mari, este posibil ca la o clasă să trebuiască să lucreze mai mulţi programatori simultan - fiecare concentrându–se pe aspecte diferite.
- când se lucrează cu cod generat automat, acesta poate fi scris separat stfel încât programatorul să nu interfereze accidental cu el. Situaţia este frecvent întâlnită în cazul aplicaţiilor de tip Windows Forms.

De exemplu, pentru o formă nou creată (numită Form1) mediul Visual Studio va scrie un fişier numit Form1.Designer.cs care conţine partea de iniţializare a controalelor şi componentelor introduse de utilizator. Partea de tratare a evenimentelor, constructori, etc este definită într-un alt fişier (Form1.cs).

Declararea unei părţi a unei clase se face folosind cuvântul cheie *partial* înaintea lui *class*.

Exemplu:

```
//fisierul Browser1.cs
public partial class Browser
{
  public void OpenPage(String uri)
  {...}
}
//fisierul Browser2.cs
public partial class Browser
{
  public void DiscardPage(String uri)
  {...}
}
```

Următoarele sunt valabile pentru tipuri parţiale:

- cuvântul *partial* trebuie să apara exact înainte cuvintelor: *class*, *interface*, *struct*
- dacă pentru o parte se specifică un anumit grad de acces, aceasta nu trebuie să ducă la conflicte cu declaraţiile din alte părţi

- dacă o parte de clasă este declarată ca abstractă, atunci întreaga clasă este considerată abstractă
- dacă o parte declară clasa ca fiind *sealed*, atunci întreaga clasă este considerată sealed
- dacă o parte declară că moşteneşte o clasă, atunci într-o altă parte nu se mai poate specifica o altă derivare
- părţi diferite pot să declare că se implementează interfeţe multiple
- aceleaşi câmpuri şi metode nu pot fi definite în mai multe părţi.
- clasele imbricate pot să fie declarate în părţi diferite, chiar dacă clasa conţinătoare e definită într-un singur fişier:

```
class Container
{
  partial class Nested
  {
    void Test() { }
  }
  partial class Nested
  {
    void Test2() { }
  }
}
```

- Următoarele elemente vor fi reunite pentru definiţia clasei: comentarii XML, interfeţe, atribute, membri.

  Exemplu:

```
partial class Earth : Planet, IRotate { }
partial class Earth : IRevolve { }
```

  este echivalent cu:

```
class Earth : Planet, IRotate, IRevolve { }
```

5.4 Structuri

Structurile reprezintă tipuri de date asemănătoare claselor, cu principala diferenţă că sunt tipuri valoare (o astfel de variabilă va conţine direct valoarea, şi nu o adresă de memorie). Sunt considerate versiuni "uşoare" ale claselor, sunt folosite predilect pentru tipuri pentru care aspectul comportamental este mai puţin pronunţat.

Declaraţia unei structuri se face astfel:
atribute$_{opt}$ modificatori-de-struct$_{opt}$ `struct` identificator :interfeţe$_{opt}$ corp ;$_{opt}$
Modificatorii de structură sunt: *new, public, protected, internal, private.* O structură este automat derivată din *System.ValueType*, care la rândul ei este derivată din *System.Object*; de asemenea, este automat considerată *sealed* (nederivabilă). Poate însă să implementeze una sau mai multe interfeţe.

O structură poate să conţină declaraţii de constante, câmpuri, metode, proprietăţi, evenimente, indexatori, operatori, constructori, constructori statici, tipuri imbricate. Nu poate conţine destructor.

La atribuire, se face o copiere a valorilor conţinute de către sursă în destinaţie (indiferent de tipul câmpurilor: valoare sau referinţă).

Exemplu:

```
using System;
public struct Point
{
  public Point(int xCoordinate, int yCoordinate)
  {
    xVal = xCoordinate;
    yVal = yCoordinate;
  }
  public int X
  {
    get
    {
      return xVal;
    }
    set
    {
      xVal = value;
    }
  }
  public int Y
  {
```

```
    get
    {
      return yVal;
    }
    set
    {
      yVal = value;
    }
}

  public override string ToString( )
  {
    return (String.Format(‘‘{0}, {1}’’, xVal,yVal));
  }
  public int xVal;
  public int yVal;
}

public class Tester
{
  public static void MyFunc(Point loc)
  {
    loc.X = 50;
    loc.Y = 100;
    Console.WriteLine(‘‘In MyFunc loc: {0}’’, loc);
  }
  static void Main( )
  {
    Point loc1 = new Point(200,300);
    Console.WriteLine(‘‘Loc1 location: {0}’’, loc1);
    MyFunc(loc1);
    Console.WriteLine(‘‘Loc1 location: {0}’’, loc1);
  }
}
```

După cum este dat în exemplul de mai sus, crearea unei instanţe se face folosind operatorul *new*; dar în acest caz, nu se va crea o instanţă în memoria heap, ci pe stivă. Transmiterea lui loc1 făcându–se prin valoare, adică metoda *myFunc* nu face decât să modifice o copie de pe stivă a lui *loc1*. La revenire, se va afişa tot valoarea originală, deoarece *loc1* a rămas nemodificat:

```
Loc1 location: 200, 300
```

```
In MyFunc loc: 50, 100
Loc1 location: 200, 300
```

Deseori pentru o structură se declară câmpurile ca fiind publice, pentru a nu mai fi necesare definirea accesorilor (simplificare implementării). Alţi programatori consideră însă că accesarea membrilor trebuie să se facă precum la clase, folosind proprietăţi. Oricare ar fi alegerea, limbajul o sprijină.

Alte aspecte demne de reţinut:

- Câmpurile nu pot fi iniţializate la declarare; altfel spus, dacă în exemplul de mai sus se scria:

  ```
  public int xVal = 10;
  public int yVal = 20;
  ```

  s-ar fi semnalat o eroare la compilare.

- Nu se poate defini un constructor implicit. Cu toate acestea, compilatorul va crea un astfel de constructor, care va iniţializa câmpurile la valorile lor implicite (0 pentru tipuri numerice sau pentru enumerări, *false* pentru *bool*, *null* pentru tipuri referinţă).

  Pentru tipul *Point* de mai sus, următoarea secvenţă de cod este corectă:

  ```
  Point a = new Point(0, 0);
  Point b = new Point();
  ```

  şi duce la crearea a două puncte cu abcisele şi ordonatele 0. Un constructor implicit este apelat atunci când se creează un tablou de structuri:

  ```
  Point[] points = new Points[10];
  for( int i=0; i<points.Length; i++ )
  {
    Console.WriteLine(points[i]);
  }
  ```

  va afişa de 10 ori puncte de coordonate (0, 0). De asemenea este apelat la iniţializarea membrilor de tip structură ai unei clase.

  De menţionat pentru exemplul anterior că se creează un obiect de tip tablou în heap, după care în interiorul lui (şi nu pe stivă!) se creează cele 10 puncte (alocare inline).

- Nu se poate declara destructor. Aceştia se declară numai pentru clase.
- Dacă programatorul defineşte un constructor, atunci acesta trebuie să dea valori iniţiale pentru câmpurile conţinute, altfel apare eroare la compilare.
- Dacă pentru instanţierea unei structuri declarate în interiorul unei metode sau bloc de instrucţiuni în general nu se apelează *new*, atunci respectiva instanţă nu va avea asociată nici o valoare (constructorul implicit nu este apelat automat!). Nu se poate folosi respectiva variabilă de tip structură decât după ce i se iniţializează toate câmpurile:

  ```
  {//bloc de instructiouni
   Point p;
   Console.WriteLine(p);
  }
  ```

  va duce la apariţia erorii de compilare:

  ```
  Use of unassigned local variable ‘p’
  ```

  Dar după nişte asignări de tipul:

  ```
  p.xVal=p.yVal=0;
  ```

  afişarea este posibilă (practic, orice apel de metodă pe instanţă este acum acceptat).
- Dacă se încearcă definirea unei structuri care conţine un câmp de tipul structurii, atunci va apărea o eroare de compilare:

  ```
  struct MyStruct
  {
    MyStruct s;
  }
  ```

  va genera un mesaj din partea compilatorului:

  ```
  Struct member ’MyStruct.s’ of type ’MyStruct’ causes a
  cycle in the structure layout
  ```

- Dacă o instanţă este folosită acolo unde un *object* este necesar, atunci se va face automat o conversie implicită către *System.Object* (boxing). Ca atare, utilizarea unei structuri poate duce (dar nu obligatoriu, ci în funcţie de context) la un overhead datorat conversiei.

5.4.1 Structuri sau clase?

Structurile pot fi mult mai eficiente în alocarea memoriei atunci când sunt reţinute într–un tablou. De exemplu, crearea unui tablou de 100 de elemente de tip *Point* (de mai sus) va duce la crearea unui singur obiect (tabloul), iar cele 100 de instanţe de tip structură ar fi alocate inline în vectorul creat (şi nu referinţe ale acestora). Dacă *Point* ar fi declarat ca şi clasă, ar fi fost necesară crearea a 101 instanţe de obiecte în heap (un obiect pentru tablou, alte 100 pentru puncte), ceea ce ar duce la mai mult lucru pentru garbage collector şi ar putea duce la fragmentarea heap-ului.

Dar în cazul în care structurile sunt folosite în colecţii de tip Object (de exemplu un *ArrayList*), se va face automat un boxing, ceea ce duce la overhead (consum suplimentar de memorie şi cicli procesor). De asemenea, la transmiterea prin valoare a unei structuri, se va face copierea tuturor câmpurilor conţinute pe stivă, ceea ce poate duce la un overhead semnificativ.

5.5 Interfeţe

O interfaţă defineşte un contract. O clasă sau o structură care implementează o interfaţă aderă la acest contract. Relaţia dintre o interfaţă şi un tip care o implementează este deosebită de cea existentă între clase (este un/o): este o relaţie de implementare.

O interfaţă conţine metode, proprietăţi, evenimente, indexatori. Ea însă nu va conţine implementări pentru acestea, doar declaraţii. Declararea unei interfeţe se face astfel:
atribute$_{opt}$ modificatori-de-interfaţă$_{opt}$ `interface` identificator baza-interfeţei$_{opt}$ corp-interfaţă ;$_{opt}$
Modificatorii de interfaţă sunt: *new, public, protected, internal, private.* O interfaţă poate să moştenească de la zero sau mai multe interfeţe. Corpul interfeţei conţine declaraţii de metode, fără implementări. Orice metodă are gradul de acces public. Nu se poate specifica pentru o metodă din interiorul unei interfeţe: *abstract, public, protected, internal, private, virtual, override*, ori *static*.

Exemplu:

```
interface IStorable
{
 void Read( );
 void Write();
}
```

O clasă care implementează o astfel de interfaţă se declară ca mai jos:

```
class Document: IStorable
{
 public void Read(){/*cod*/}
 public void Write(){/*cod*/}
 //alte declaratii
}
```

O clasă care implementează o interfaţă trebuie să definească toate metodele care se regăsesc în interfaţa respectivă, sau să declare metodele din interfaţă ca fiind abstracte. Următoarele reguli trebuie respectate la implementarea de interfeţe:

1. Tipul de retur al metodei din clasă trebuie să coincidă cu tipul de retur al metodei din interfaţă
2. Tipul parametrilor formali din metodă trebuie să fie acelaşi cu tipul parametrilor formali din interfaţă
3. Metoda din clasă trebuie să fie declarată publică şi nestatică.

Aceste implementări pot fi declarate folosind specificatorul *virtual* (deci subclasele clasei curente pot folosi *new* şi *override*).

Exemplu:

```
using System;
interface ISavable
{
 void Read();
 void Write();
}
public class TextFile : ISavable
{
 public virtual void Read()
 {
  Console.WriteLine("TextFile.Read()");
 }
 public void Write()
 {
  Console.WriteLine("TextFile.Write()");
 }
}
public class ZipFile : TextFile
{
```

```
 public override void Read()
 {
  Console.WriteLine("ZipFile.Read()");
 }
 public new void Write()
 {
  Console.WriteLine("ZipFile.Write()");
 }
}
public class Test
{
 static void Main()
 {
  Console.WriteLine("\nTextFile reference to ZipFile");
  TextFile textRef = new ZipFile();
  textRef.Read();
  textRef.Write();
  Console.WriteLine("\nISavable reference to ZipFile");
  ISavable savableRef = textRef as ISavable;
  if(savableRef != null)
  {
   savableRef.Read();
   savableRef.Write();
  }
  Console.WriteLine("\nZipFile reference to ZipFile");
  ZipFile zipRef = textRef as ZipFile;
  if(zipRef!= null)
  {
   zipRef.Read();
   zipRef.Write();
  }
 }
}
```

La ieşire se va afişa:

```
TextFile reference to ZipFile
ZipFile.Read()
TextFile.Write()

ISavable reference to ZipFile
ZipFile.Read()
```

```
TextFile.Write()

ZipFile reference to ZipFile
ZipFile.Read()
ZipFile.Write()
```

În exemplul de mai sus se foloşeste operatorul *as* pentru a obţine o referinţă la interfeţe, pe baza obiectelor create. În general, se preferă ca apelul metodelor care sunt implementate din interfaţă să se facă via o referinţă la interfaţa respectivă, obţinută prin intermediul operatorului *as* (ca mai sus) sau după o testare prealabilă prin *is* urmată de conversie explicită, ca mai jos:

```
if (textRef is ISavable)
{
 ISavable is = (ISavable)textRef;
 is.Read();//etc
}
```

În general, dacă se doreşte doar răspunsul la întrebarea "este obiectul curent un implementator al interfeţei I?", atunci se recomandă folosirea operatorului *is*. Dacă se ştie că va trebui făcută şi o conversie la tipul interfaţă, atunci este mai eficientă folosirea lui *as*. Afirmaţia se bazează pe studiul codului IL rezultat în fiecare caz.

Să presupunem că exista o interfaţă `I` având metoda `M()` care este implementată de o clasă `C`, care defineşte metoda `M()`. Este posibil ca această metodă să nu aibă o semnificaţie în afara clasei `C`, ca atare a e de dorit ca metoda `M()` să nu fie declarată publică. Mecanismul care permite acest lucru se numeşte *implementare explicită*. Această tehnică permite ascunderea metodelor moştenite dintr-o interfaţă, acestea devenind private (calificarea lor ca fiind publice este semnalată ca o eroare). Implementarea explicită se obţine prin calificarea numelui de metodă cu numele interefţei:

```
interface IMyInterface
{
void F();
}
class MyClass : IMyInterface
{
void IMyInterface.F()
{
//...
}
}
```

Metodele din interfeţe care s–au implementat explicit nu pot fi declarate *abstract, virtual, override, new.* Mai mult, asemenea metode nu pot fi accesate direct prin intermediul unui obiect (*obiect.NumeMetoda*), ci doar prin intermediul unei conversii către interfaţă respectivă, deoarece prin implementare explicită a metodelor aceste devin private şi singura modalitate de acces a lor este upcasting-ul către interfaţă.

Exemplu:

```
using System;
public interface IDataBound
{
 void Bind();
}

public class EditBox : IDataBound
{
 // implementare de IDataBound
 void IDataBound.Bind()
 {
  Console.WriteLine("Binding to data store...");
 }
}

class NameHidingApp
{
 public static void Main()
 {
  Console.WriteLine();
  EditBox edit = new EditBox();
  Console.WriteLine("Apel EditBox.Bind()...");
  //EROARE: Aceasta linie nu se compileaza deoarece metoda
  //Bind nu mai exista ca metoda publica in clasa EditBox
  edit.Bind();
  Console.WriteLine();

  IDataBound bound = edit;
  Console.WriteLine("Apel (IDataBound) EditBox.Bind()...");
  // Functioneaza deoarece s-a facut conversie la IDataBound
  bound.Bind();
 }
}
```

Este posibil ca un tip să implementeze mai multe interfeţe. Atunci când două interfeţe au o metodă cu aceeaşi semnătură, programatorul are mai multe variante de lucru. Cel mai simplu, el poate să furnizeze o singură implementare pentru ambele metode, ca mai jos:

```
interface IFriendly
{
 void GreetOwner() ;
}
interface IAffectionate
{
 void GreetOwner() ;
}
abstract class Pet
{
 public virtual void Eat()
 {
  Console.WriteLine( "Pet.Eat" ) ;
 }
}
class Dog : Pet, IAffectionate, IFriendly
{
 public override void Eat()
 {
  Console.WriteLine( "Dog.Eat" ) ;
 }
 public void GreetOwner()
 {
  Console.WriteLine( "Woof!" ) ;
 }
}
```

O altă modalitate este să se specifice implicit care metodă este implementată.

```
class Dog : Pet, IAffectionate, IFriendly
{
 public override void Eat()
 {
  Console.WriteLine( "Dog.Eat" ) ;
 }
 void IAffectionate.GreetOwner()
 {
```

```
  Console.WriteLine( "Woof!" ) ;
 }
 void IFriendly.GreetOwner()
 {
  Console.WriteLine( "Jump up!" ) ;
}
}
public class Pets
{
  static void Main()
  {
    IFriendly mansBestFriend = new Dog() ;
    mansBestFriend.GreetOwner() ;
    (mansBestFriend as IAffectionate).GreetOwner() ;
  }
}
```

La ieşire se va afişa:

```
Jump up!
Woof!
```

Dacă însă în clasa Dog se adaugă metoda

```
public void GreetOwner()
{
Console.WriteLine( "Woof!" ) ;
}
```

atunci se poate face apel de tipul `dog.GreetOwner()` (variabila dog este instanţă de Dog); apelurile de metode din interfaţă rămân de asemenea valide. Rezultatul este afişarea mesajului `Woof`.

5.5.1 Clase abstracte sau interfeţe?

Atât interfeţele cât şi clasele abstracte au comportamente similare şi pot fi folosite în situaţii similare. Dar totuşi ele nu se pot substitui reciproc. Câteva principii generale de utilizare a lor sunt date mai jos.

Dacă o relaţie se poate exprima mai degrabă ca "este un/o" decât altfel, atunci entitatea de bază ar trebui gândită ca o clasă abstractă.

Un alt aspect este bazat pe obiectele care ar folosi capabilităţile din tipul de bază. Dacă aceste capabilităţi ar fi folosite de către obiecte care nu sunt legate între ele, atunci ar fi indicată o interfaţă.

Dezavantajul claselor abstracte este că nu poate fi decât bază unică pentru orice altă clasă.

5.6 Tipul delegat

În programare deseori apare următoarea situaţie: trebuie să se execute o anumită acţiune, dar nu se ştie de dinainte care anume, sau chiar ce obiect va trebui efectiv utilizat. De exemplu, un buton poate şti că trebuie să anunţe pe oricine este intresat despre faptul că fost apăsat, dar nu va şti aprioric cum va fi tratat acest eveniment. Mai degrabă decât să se lege butonul de un obiect particular, butonul va declara un delegat, pentru care clasa interesată de evenimentul de apăsare va da o implementare.

Fiecare acţiune pe care utilizatorul o execută pe o interfaţă grafică declanşează un eveniment. Alte evenimente se pot declanşa independent de acţiunile utilizatorului: sosirea unui email, terminarea copierii unor fişiere, sfârşitul unei interogări pe o bază de date, etc. Un eveniment este o încapsulare a ideii că "se întâmplă ceva" la care programul trebuie să răspundă. Evenimentele şi delegaţii sunt strâns legate deoarece răspunsul la acest eveniment se va face de către un event handler, care este legat de eveniment printr-un delegat,

Un delegat este un tip referinţă folosit pentru a încapsula o metodă cu un anumit antet (tipul parametrilor formali şi tipul de retur). Orice metodă care are acest antet poate fi legată la un anumit delegat. Într–un limbaj precum C++, acest lucru se rezolvă prin intermediul pointerilor la funţii. Delegaţii rezolvă aceeaşi problemă, dar într–o manieră orientată obiect şi cu garanţii asupra siguranţei codului rezultat, precum şi cu o uşoară generalizare (vezi delegaţii multicast).

Un delegat este creat după următoarea sintaxă:
atribute$_{opt}$ modificatori-de-delegat$_{opt}$ `delegate` tip-retur identificator(lista-param-formali$_{opt}$);
Modificatorul de delegat poate fi: *new, public, protected, internal, private.* Un delegat se poate specifica atât în interiorul unei clase, cât şi în exteriorul ei, fiind de fapt o declaraţie de clasă derivată din *System.Delegate.* Dacă este declarat în interiorul unei clase, atunci este şi static (asemănător cu statutul claselor imbricate).

Exemplu:

```
public delegate int WhichIsFirst(object obj1, object obj2);
```

5.6.1 Utilizarea delegaţilor pentru a specifica metode la runtime

Să presupunem că se doreşte crearea unei clase container simplu numit *Pair* care va conţine două obiecte pentru care va putea face şi sortare. Nu se va şti aprioric care va fi tipul obiectelor conţinute, deci se va folosi pentru ele tipul *object*. Dar sortarea celor două obiecte se va face diferit, în funcţie de tipul lor efectiv: de exemplu pentru nişte persoane (clasa *Student* în cele ce urmează) se va face după nume, pe când pentru animale (clasa *Dog*) se va face după alt criteriu: greutatea. Containerul *Pair* va trebui să facă faţă acestor clase diferite. Rezolvarea se va da prin delegaţi.

Clasa *Pair* va defini un delegat, *WhichIsFirst*. Metoda *Sort* de ordonare va prelua ca (unic) parametru o instanţă a metodei *WhichIsFirst*, care va implementa relaţia de ordine, în funcţie de tipul obiectelor conţinute. Rezultatul unei comparaţii între două obiecte va fi de tipul enumerare *Comparison*, definit de utilizator:

```
public enum Comparison
{
  theFirstComesFirst = 0,
  //primul obiect din colectie este primul in ordinea sortarii
  theSecondComesFirst = 1
  //al doilea obiect din colectie este primul in ordinea sortarii
}
```

Delegatul (tipul de metodă care realizează compararea) se declară astfel:

```
//declarare de delegat
public delegate Comparison WhichIsFirst(
                          object obj1, object obj2);
```

Clasa Pair se declară după cum urmează:

```
public class Pair
{
  //tabloul care contine cele doua obiecte
  private object[] thePair = new object[2];

  //constructorul primeste cele doua obiecte continute
  public Pair( object firstObject, object secondObject)
  {
    thePair[0] = firstObject;
    thePair[1] = secondObject;
```

```
  }

  //metoda publica pentru ordonarea celor doua obiecte
  //dupa orice criteriu
  public void Sort( WhichIsFirst theDelegatedFunc )
  {
    if (theDelegatedFunc(thePair[0],thePair[1]) ==
        Comparison.theSecondComesFirst)
    {
      object temp = thePair[0];
      thePair[0] = thePair[1];
      thePair[1] = temp;
    }
  }
  //metoda ce permite tiparirea perechii curente
  //se foloseste de polimorfism - vezi mai jos
  public override string ToString( )
  {
    return thePair[0].ToString()+", "+thePair[1].ToString();
  }
}
```

Clasele Student şi Dog sunt:

```
public class Dog
{
  public Dog(int weight)
  {
    this.weight=weight;
  }
  //Ordinea este data de greutate
  public static Comparison WhichDogComesFirst(
    Object o1, Object o2)
  {
    Dog d1 = o1 as Dog;
    Dog d2 = o2 as Dog;
    return d1.weight > d2.weight ?
     Comparison.theSecondComesFirst :
     Comparison.theFirstComesFirst;
  }
  //pentru afisarea greutatii unui caine
  public override string ToString( )
```

```
  {
    return weight.ToString( );
  }
  private int weight;
}

public class Student
{
  public Student(string name)
  {
    this.name = name;
  }
  //studentii sunt ordonati alfabetic
  public static Comparison WhichStudentComesFirst(
    Object o1, Object o2)
  {
    Student s1 = o1 as Student;
    Student s2 = o2 as Student;
    return (String.Compare(s1.name, s2.name) < 0 ?
      Comparison.theFirstComesFirst :
      Comparison.theSecondComesFirst);
  }
  //pentru afisarea numelui unui student
  public override string ToString( )
  {
    return name;
  }
  private string name;
}
```

Clasa de test este:

```
public class Test
{
  public static void Main( )
  {
    //creaza cate doua obiecte
    //de tip Student si Dog
    //si containerii  corespunzatori
    Student Stacey = new Student(‘‘Stacey’’);
    Student Jesse = new Student (‘‘Jess’’);
    Dog Milo = new Dog(10);
```

```
    Dog Fred = new Dog(5);
    Pair studentPair = new Pair(Stacey, Jesse);
    Pair dogPair = new Pair(Milo, Fred);
    Console.WriteLine(‘‘studentPair\t: {0}’’, studentPair);
    Console.WriteLine(‘‘dogPair\t: {0}’’, dogPair);
    //Instantiaza delegatii
    WhichIsFirst theStudentDelegate =
     new WhichIsFirst(Student.WhichStudentComesFirst);
    WhichIsFirst theDogDelegate =
     new WhichIsFirst(Dog.WhichDogComesFirst);
    //sortare folosind delegatii
    studentPair.Sort(theStudentDelegate);
    Console.WriteLine(‘‘Dupa sortarea pe studentPair\t: {0}’’,
     studentPair.ToString( ));
    dogPair.Sort(theDogDelegate);
    Console.WriteLine(‘‘Dupa sortarea pe dogPair\t\t: {0}’’,
     dogPair.ToString( ));
  }
}
```

5.6.2 Delegaţi statici

Unul din aspectele neelegante ale exemplului anterior este că e necesar ca în clasa *Test* să se instanţieze delegaţii care sunt necesari pentru a ordona obiectele din *Pair*. O modalitate mai bună este să se obţină delegaţii direct din clasele *Student* şi *Dog*. Acest lucru se obţine prin crearea unui delegat static în interiorul fiecărei clase:

```
public static readonly WhichIsFirst OrderStudents =
new WhichIsFirst(Student.WhichStudentComesFirst);
```

(analog pentru clasa *Dog*; de remarcat că “static readonly” nu se poate înlocui cu “const”, deoarece iniţilizatorul nu este considerat expresie constantă). Declaraţia de mai sus se foloseşte astfel:

```
...
studentpair.Sort(Student.OrderStudent);
...
```

rezultatul fiind identic.

În [2] este dată şi o implementare de delegat ca proprietate statică, aceasta ducând la crearea unui delegat doar în cazul în care este nevoie de el[1].

[1] Tehnică numită iniţializaer târzie (lazy initialization)

5.6.3 Multicasting

Uneori este nevoie ca un delegat să poată apela mai mult de o singură metodă. De exemplu, atunci când un buton este apăsat, se poate să vrei să efectuezi mai mult de o sigură acţiune: să scrii într–un textbox un şir de caractere şi să înregistrezi într–un fişier faptul că s–a apăsat acel buton (logging). Acest lucru s–ar putea rezolva prin construirea unui vector de delegaţi care să conţină toate metodele dorite, însă s-ar ajunge la un cod greu de urmărit şi inflexibil; pentru un astfel de exemplu, a se vedea [2]. Mult mai simplu ar fi dacă unui delegat i s-ar putea atribui mai multe metode. Acest lucru se numeşte *multicasting* şi este folosit intens la tratarea evenimentelor.

Orice delegat care returnează *void* este un delegat multicast, care poate fi tratat şi ca un delegat single-cast. Doi delegaţi multicast pot fi combinaţi folosind semnul +. Rezultatul unei astfel de "adunări" este un nou delegat multicast care la apelare va invoca metodele conţinute, în ordinea în care s–a făcut adunarea. De exemplu, dacă *Writer* şi *Logger* sunt delegaţi care returnează void, atunci următoarea linie va produce combinarea lor într–un singur delegat:

```
myMulticastDelegate = Writer + Logger;
```

Se pot adăuga delegaţi multicast folosind operatorul +=, care va adăuga delegatul de la dreapta operatorului la delegatul multicast aflat în stânga sa:

```
myMulticastDelegate += Transmitter;
```

presupunând că Transmitter este compatibil cu *myMulticastDelegate* (are aceeaşi semnătură). Operatorul − = funcţionează invers faţă de + =.

Exemplu:

```
using System;
//declaratia de delegat multicast
public delegate void StringDelegate(string s);

public class MyImplementingClass
{
  public static void WriteString(string s)
  {
    Console.WriteLine("Writing string {0}", s);
  }
  public static void LogString(string s)
  {
    Console.WriteLine("Logging string {0}", s);
```

```
  }
  public static void TransmitString(string s)
  {
    Console.WriteLine("Transmitting string {0}", s);
  }
}

public class Test
{
  public static void Main( )
  {
    //defineste trei obiecte delegat
    StringDelegate Writer,
      Logger, Transmitter;
    //defineste alt delegat
    //care va actiona ca un delegat multicast
    StringDelegate myMulticastDelegate;
    //Instantiaza primii trei delegati
    //dand metodele ce se vor incapsula
    Writer = new StringDelegate(
      MyImplementingClass.WriteString);
    Logger = new StringDelegate(
      MyImplementingClass.LogString);
    Transmitter = new StringDelegate(
      MyImplementingClass.TransmitString);
    //Invoca metoda delegat Writer
    Writer("String passed to Writer\n");
    //Invoca metoda delegat Logger
    Logger("String passed to Logger\n");
    //Invoca metoda delegat Transmitter
    Transmitter("String passed to Transmitter\n");
    //anunta utilizatorul ca va combina doi delegati
    Console.WriteLine(
      "myMulticastDelegate = Writer + Logger");
    //combina doi delegati, rezultatul este
    //asignat lui myMulticastDelagate
    myMulticastDelegate = Writer + Logger;
    //apelaeaza myMulticastDelegate
    //de fapt vor fi chemate cele doua metode
    myMulticastDelegate(
      "First string passed to Collector");
```

```
    //Anunta utilizatorul ca se va adauga al treilea delegat
    Console.WriteLine(
      "\nmyMulticastDelegate += Transmitter");
    //adauga al treilea delegat
    myMulticastDelegate += Transmitter;
    //invoca cele trei metode delagate
    myMulticastDelegate(
      "Second string passed to Collector");
    //anunta utilizatorul ca se va scoate delegatul Logger
    Console.WriteLine(
      "\nmyMulticastDelegate -= Logger");
    //scoate delegatul Logger
    myMulticastDelegate -= Logger;
    //invoca cele doua metode delegat ramase
    myMulticastDelegate(
      "Third string passed to Collector");
  }
}
```

La ieşire vom avea:

```
Writing string String passed to Writer
Logging string String passed to Logger
Transmitting string String passed to Transmitter
myMulticastDelegate = Writer + Logger
Writing string First string passed to Collector
Logging string First string passed to Collector
myMulticastDelegate += Transmitter
Writing string Second string passed to Collector
Logging string Second string passed to Collector
Transmitting string Second string passed to Collector
myMulticastDelegate -= Logger
Writing string Third string passed to Collector
Transmitting string Third string passed to Collector
```

Curs 6

Metode anonime. Evenimente. Excepţii.

6.1 Metode anonime

Pentru a folosi un delegat a fost nevoie până acum de a se crea de fiecare dată o metodă (şi posibil şi o nouă clasă care să conţină această metodă). Există însă cazuri în care corpul metodei este suficient de simplu pentru a nu necesita declararea explicită a metodei. C# 2.0 introduce această facilitate prin intermediul metodelor anonime.

Varianta tradiţională presupunea scrierea unui cod de forma:

```
class SomeClass
{
   delegate void SomeDelegate();
   public void InvokeMethod()
   {
      SomeDelegate del = new SomeDelegate(SomeMethod);
      del();
   }
   void SomeMethod()
   {
      MessageBox.Show("Hello");
   }
}
```

Se poate defini o implementare folosind o metodă anonimă:

```
class SomeClass
{
```

```
   delegate void SomeDelegate();
   public void InvokeMethod()
   {
      SomeDelegate del = delegate()
                         {
                             MessageBox.Show("Hello");
                         };
      del();
   }
}
```

Metoda anonimă este definită in-line şi nu ca metodă membră a unei clase. Compilatorul este suficient de inteligent pentru a infera tipul delegatului, pe baza declaraţiei de variabilă delegat (del în exemplul anterior).

O astfel de metodă anonimă se poate folosi oriunde este nevoie de o variabilă de tip delegat, de exemplu ca parametru al unei metode:

```
class SomeClass
{
   delegate void SomeDelegate();
   public void SomeMethod()
   {
      InvokeDelegate(delegate(){MessageBox.Show("Hello");});
   }
   void InvokeDelegate(SomeDelegate del)
   {
      del();
   }
}
```

Există şi cazuri în care se cere transmiterea de parametri metodei anonime. Parametrii (tip + nume) se declară în interiorul parantezelor cuvântului `delegate`:

```
class SomeClass
{
   delegate void SomeDelegate(string str);
   public void InvokeMethod()
   {
      SomeDelegate del = delegate(string str)
                         {
                             MessageBox.Show(str);
```

```
                        };
      del("Hello");
   }
}
```

Dacă se omit cu totul parantezele de după cuvântul `delegate`, atunci se declară o metodă anonimă care este asignabilă unui delegat cu orice semnătură:

```
class SomeClass
{
   delegate void SomeDelegate(string str);
   public void InvokeMethod()
   {
      SomeDelegate del = delegate
                        {
                            MessageBox.Show("Hello");
                        };
      del("Parameter is ignored");
   }
}
```

Remarcăm că încă trebuie daţi parametri delegatului ce se apelează; dacă delegatul declară parametri de tip out, atunci varianta de mai sus nu se poate aplica.

6.2 Evenimente

Interfeţele grafice actuale cer ca un anumit program să răspundă la *evenimente*. Un eveniment poate fi de exemplu apăsarea unui buton, terminarea transferului unui fişier, selectarea unui meniu, etc; pe scurt, se întâmplă ceva la care trebuie să se dea un răspuns. Nu se poate prezice ordinea în care se petrec evenimentele, iar la apariţia unuia se va cere reacţionarea din partea sistemului soft.

Alte clase pot fi interesate în a răspunde la aceste evenimente. Modul în care vor reacţiona va fi extrem de particular, iar obiectul care semnalează evenimentul (ex: un obiect de tip buton, la apăsarea lui) nu trebuie să ştie modul în care se va răspunde. Butonul va comunica faptul că a fost apăsat, iar clasele interesate în acest eveniment vor reacţiona în consecinţă.

6.2.1 Publicarea şi subscrierea

În C#, orice obiect poate să *publice* un set de evenimente la care alte clase pot să *subscrie*. Când obiectul care a publicat evenimentul îl şi semnalează, toate obiectele care au subscris la acest eveniment sunt notificate. În acest mod se defineşte o dependenţă de tip *one–to–many* între obiecte astfel încât dacă un obiect îşi schimbă starea, atunci toate celelate obiecte dependente sunt notificate şi modificate automat.

De exemplu, un buton poate să notifice un număr oarecare de observatori atunci când a fost apăsat. Butonul va fi numit *publicator*[1] deoarece publică evenimentul *Click* iar celelalte clase sunt numite *abonaţi*[2] deoarece ele subscriu la evenimentul Click.

6.2.2 Evenimente şi delegaţi

Tratarea evenimentelor în C# se face folosind delegaţi. Clasa ce publică defineşte un delegat pe care clasele abonate trebuie să îl implementeze. Când evenimentul este declanşat, metodele claselor abonate vor fi apelate prin intermediul delegatului (pentru care se prevede posibilitatea de a fi multicast, astfel încât să se permită mai mulţi abonaţi).

Metodele care răspund la un eveniment se numesc *event handlers*. Prin convenţie, un event handler în .NET Framework returnează *void* şi preia doi parametri: primul parametru este sursa evenimentului (obiectul publicator); al doilea parametru are tip *EventArgs* sau derivat din acesta.

Declararea unui eveniment se face astfel:
atribute$_{opt}$ modificatori-de-eveniment$_{opt}$ `event` tip nume–eveniment
Modificator-de-eveniment poate fi *abstract, new, public, protected, internal, private, static, virtual, sealed, override, extern.* *Tip* este un handler de eveniment (delegat multicast).

Exemplu:

```
public event SecondChangeHandler OnSecondChange;
```

Vom da mai jos un exemplu care va construi următoarele: o clasă *Clock* care foloseşte un eveniment (*OnSecondChange*) pentru a notifica potenţialii abonaţi atunci când timpul local se schimbă cu o secundă. Tipul acestui eveniment este un delegat *SecondChangeHandler* care se declară astfel:

```
public delegate void SecondChangeHandler(
  object clock, TimeInfoEventArgs timeInformation );
```

[1] Engl: publisher
[2] Engl: subscribers

în conformitate cu metodologia de declarare a unui event handler, pomenită mai sus. Tipul *TimeInfoEventArgs* este definit de noi ca o clasă derivată din *EventArgs*:

```
public class TimeInfoEventArgs : EventArgs
{
  public TimeInfoEventArgs( int hour, int minute, int second )
  {
    this.hour = hour;
    this.minute = minute;
    this.second = second;
  }
  public readonly int hour;
  public readonly int minute;
  public readonly int second;
}
```

Această clasă va conţine informaţie despre timpul curent. Informaţia este accesibilă *readonly*.

Clasa *Clock* va conţine o metodă *Run()*:

```
public void Run()
{
  for(;;)
  {
    //dormi 10 milisecunde
    Thread.Sleep(10);
    //obtine timpul curent
    System.DateTime dt = System.DateTime.Now();
    //daca timpul s-a schimbat cu o secunda
    //atunci notifica abonatii
    if( dt.Second != second)
    //second este camp al clasei Clock
    {
      //creeaza obiect TimeInfoEventArgs
      //ce va fi transmis abonatilor
      TimeInfoEventArgs timeInformation =
        new TimeInfoEventArgs(dt.Hour, dt.Minute, dt.Second);
      //daca cineva este abonat, atunci anunta-l
      if (OnSecondChange != null)
      {
        OnSeconChange(this, timeInformation);
```

```
      }
    }
    //modifica timpul curent in obiectul Clock
    this.second = dt.Second;
    this.minute = dt.Minute;
    this.hour = dt.Hour;
  }
}
```

Metoda *Run* creează un ciclu infinit care interoghează periodic ceasul sistem. Dacă timpul s–a schimbat cu o secundă faţă de timpul precedent, se vor notifica toate obiectele abonate după care îşi va modifica starea, prin cele trei atribuiri finale.

Tot ce rămâne de făcut este să se scrie nişte clase care să subscrie la evenimentul publicat de clasa *Clock*. Vor fi două clase: una numită *DisplayClock* care va afişa pe ecran timpul curent şi o alta numită *LogCurrentTime* care ar trebui să înregistreze evenimentul într–un fişier, dar pentru simplitate va afişa doar la dispozitivul curent de ieşire informaţia transmisă:

```
public class DisplayClock
{
  public void Subscribe(Clock theClock)
  {
    theClock.OnSecondChange +=
      new Clock.SecondChangeHandler(TimeHasChanged);
  }
  void TimeHasChanged(
    object theClock, TimeInfoEventArgs ti)
  {
    Console.WriteLine("Current Time: {0}:{1}:{2}",
      ti.hour.ToString( ),
      ti.minute.ToString( ),
      ti.second.ToString( ));
  }
}
public class LogCurrentTime
{
  public void Subscribe(Clock theClock)
  {
    theClock.OnSecondChange +=
      new Clock.SecondChangeHandler(WriteLogEntry);
  }
```

```
  //Aceasta metoda ar trebui sa scrie intr-un fisier
  //dar noi vom scrie la consola
  void WriteLogEntry(
    object theClock, TimeInfoEventArgs ti)
  {
    Console.WriteLine("Logging to file: {0}:{1}:{2}",
    ti.hour.ToString( ),
    ti.minute.ToString( ),
    ti.second.ToString( ));
  }
}
```

De remarcat faptul că evenimentele sunt adăugate folosind operatorul +=.

Exemplul în întregime este dat mai jos:

```
using System;
using System.Threading;
//o clasa care va contine informatie despre eveniment
//in acest caz va contine informatie disponibila in clasa Clock
public class TimeInfoEventArgs : EventArgs
{
  public TimeInfoEventArgs(int hour, int minute, int second)
  {
    this.hour = hour;
    this.minute = minute;
    this.second = second;
  }
  public readonly int hour;
  public readonly int minute;
  public readonly int second;
}
//clasa care publica un eveniment: OnSecondChange
//clasele care se aboneaza vor subscrie la acest eveniment
public class Clock
{
  //delegatul pe care abonatii trebuie sa il implementeze
  public delegate void SecondChangeHandler(
    object clock, TimeInfoEventArgs timeInformation );
  //evenimentul ce se publica
  public event SecondChangeHandler OnSecondChange;
  //ceasul este pornit si merge la infinit
  //va declansa un eveniment pentru fiecare secunda trecuta
```

```
  public void Run( )
  {
    for(;;)
    {
      //inactiv 10 ms
      Thread.Sleep(10);
      //citeste timpul curent al sistemului
      System.DateTime dt = System.DateTime.Now;
      //daca s-a schimbat fata de secunda anterior inregistrata
      //atunci notifica pe abonati
      if (dt.Second != second)
      {
        //creaza obiectul TimeInfoEventArgs
        //care va fi transmis fiecarui abonat
        TimeInfoEventArgs timeInformation =
          new TimeInfoEventArgs(
            dt.Hour,dt.Minute,dt.Second);
        //daca cineva a subscris la acest eveniment
        //atunci anunta-l
        if (OnSecondChange != null)
        {
          OnSecondChange(this,timeInformation);
        }
      }
      //modifica starea curenta
      this.second = dt.Second;
      this.minute = dt.Minute;
      this.hour = dt.Hour;
    }
  }
  private int hour;
  private int minute;
  private int second;
}
//un observator (abonat)
//DisplayClock va subscrie la evenimentul lui Clock
//DisplayClock va afisa timpul curent
public class DisplayClock
{
  //dandu-se un obiect clock, va subscrie
  //la evenimentul acestuia
```

```
  public void Subscribe(Clock theClock)
  {
    theClock.OnSecondChange +=
      new Clock.SecondChangeHandler(TimeHasChanged);
  }
  //handlerul de eveniment de pe partea
  //clasei DisplayClock
  void TimeHasChanged(
    object theClock, TimeInfoEventArgs ti)
  {
    Console.WriteLine("Current Time: {0}:{1}:{2}",
    ti.hour.ToString( ),
    ti.minute.ToString( ),
    ti.second.ToString( ));
  }
}
//un al doilea abonat care ar trebui sa scrie intr-un fisier
public class LogCurrentTime
{
  public void Subscribe(Clock theClock)
  {
    theClock.OnSecondChange +=
    new Clock.SecondChangeHandler(WriteLogEntry);
  }
  //acest handler ar trebui sa scrie intr-un fisier
  //dar va scrie la standard output
  void WriteLogEntry(
    object theClock, TimeInfoEventArgs ti)
  {
    Console.WriteLine("Logging to file: {0}:{1}:{2}",
    ti.hour.ToString( ),
    ti.minute.ToString( ),
    ti.second.ToString( ));
  }
}
public class Test
{
  static void Main( )
  {
    //creaza un obiect de tip Clock
    Clock theClock = new Clock( );
```

```
    //creaza un obiect DisplayClock care
    //va subscrie la evenimentul obiectului
    //Clock anterior creat
    DisplayClock dc = new DisplayClock( );
    dc.Subscribe(theClock);
    //analog se creeaza un obiect de tip LogCurrentTime
    //care va subscrie la acelasi eveniment
    //ca si obiectul DisplayClock
    LogCurrentTime lct = new LogCurrentTime( );
    lct.Subscribe(theClock);
    //porneste ceasul
    theClock.Run( );
  }
}
```

La ieşire se va afişa:

```
Current Time: 14:53:56
Logging to file: 14:53:56
Current Time: 14:53:57
Logging to file: 14:53:57
Current Time: 14:53:58
Logging to file: 14:53:58
Current Time: 14:53:59
Logging to file: 14:53:59
```

6.2.3 Comentarii

S–ar putea pune următoarea întrebare: de ce este nevoie de o astfel de redirectare de eveniment, când în metoda *Run()* se poate afişa direct pe ecran sau într–un fişier informaţia cerută? Avantajul abordării anterioare este că se pot crea oricâte clase care să fie notificate atunci când acest eveniment se declanşează. Clasele abonate nu trebuie să ştie despre modul în care lucrează clasa *Clock*, iar clasa *Clock* nu trebuie să ştie despre clasele care vor subscrie la evenimentul său. Similar, un buton poate să publice un eveniment *OnClick* şi orice număr de obiecte pot subscrie la acest eveniment, primind o notificare atunci când butonul este apăsat.

Publicatorul şi abonaţii sunt decuplaţi. Clasa *Clock* poate să modifice modalitatea de detectare a schimbării de timp fără ca acest lucru să impună o schimbare în clasele abonate. De asemenea, clasele abonate pot să îşi modifice modul de tratare a evenimentului, în mod transparent faţă de clasa *Clock*. Toate aceste caracteristici fac întreţinerea codului extrem de facilă.

6.3 Tratarea excepţiilor

C#, la fel ca alte limbaje, permite tratarea erorilor şi a situaţiilor deosebite prin excepţii. O excepţie este un obiect care încapsulează informaţie despre o situaţie anormală. Ea este folosită pentru a semnala contextul în care apare situaţia deosebită

Un programator nu trebuie să confunde tratarea excepţiilor cu erorile sau bug–urile. Un bug este o eroare de programare care ar trebui să fie fixată înainte de livrarea codului. Excepţiile nu sunt gândite pentru a preveni bug–urile (cu toate că un bug poate să ducă la apariţia unei excepţii), pentru că acestea din urmă ar trebui să fie eliminate.

Chiar dacă se scot toate bug–urile, vor exista erori predictibile dar neprevenibile, precum deschiderea unui fişier al cărui nume este greşit sau împărţiri la 0. Nu se pot preveni astfel de situaţii, dar se pot manipula astfel încât nu vor duce la prăbuşirea programului. Când o metodă întâlneşte o situaţie excepţională, atunci se va arunca o excepţie; cineva va trebui să sesizeze (să "prindă") această excepţie, sau eventual să lase o funcţie de nivel superior să o trateze. Dacă nimeni nu tratează această excepţie, atunci CLR o va face, dar aceasta duce la oprirea *thread*–ului[3].

6.3.1 Tipul *Exception*

În C# se pot arunca ca excepţii obiecte de tip *System.Exception* sau derivate ale acestuia. Există o ierarhie de excepţii care se pot folosi, sau se pot crea propriile tipuri excepţie.

Enumerăm următoarele metode şi proprietăţi relevante ale clasei *Exception*:

- *public Exception(), public Exception(string), public Exception(string, Exception)* - constructori; ultimul preia un obiect de tip *Exception* (sau de tip clasă derivată) care va fi încapsulat în instanţa curentă; o excepţie poate deci să conţină în interiorul său o instanţă a altei excepţii (cea care a fost de fapt semnalată iniţial).

- *public virtual string HelpLink {get; set;}* obţine sau setează o legătură către un fişier help asociat acestei excepţii; poate fi de asemenea o adresa Web (URL)

- *public Exception InnerException {get;}* returnează excepţia care este încorporată în excepţia curentă

[3] Şi nu neapărat a întregului proces!

- *public virtual string Message {get;}* obţine un mesaj care descrie excepţia curentă
- *public virtual string Source {get; set;}* obţine sau setează numele aplicaţiei sau al obiectului care a cauzat eroarea
- *public virtual string StackTrace {get;}* obţine o reprezetare string a apelurilor de metode care au dus la apariţia acestei excepţii
- *public MethodBase TargetSite {get;}* obţine metoda care a aruncat excepţia curentă[4]

6.3.2 Aruncarea şi prinderea excepţiilor

Aruncarea cu *throw*

Aruncarea unei excepţii se face folosind instrucţiunea *throw*. Exemplu:

```
throw new System.Exception();
```

Aruncarea unei excepţii opreşte execuţia metodei curente, după care CLR începe să caute un manipulator de excepţie. Dacă un handler de excepţie nu este găsit în metoda curentă, atunci CLR va curăţa stiva, ajungându–se la metoda apelantă. Fie undeva în lanţul de metode care au fost apelate se găseşte un exception handler, fie thread–ul curent este terminat de către CLR.

Exemplu:

```
using System;
public class Test
{
  public static void Main( )
  {
    Console.WriteLine(‘‘Enter Main...’’);
    Test t = new Test( );
    t.Func1( );
    Console.WriteLine(‘‘Exit Main...’’);
  }
  public void Func1( )
  {
    Console.WriteLine(‘‘Enter Func1...’’);
```

[4]MethodBase este o clasă care pune la dispoziţie informaţii despre metodele şi constructorii unei clase

```
    Func2( );
    Console.WriteLine(‘‘Exit Func1...’’);
  }
  public void Func2( )
  {
    Console.WriteLine(‘‘Enter Func2...’’);
    throw new System.Exception( );
    Console.WriteLine(‘‘Exit Func2...’’);
  }
}
```

Se exemplifică apelul de metode: *Main()* apelează *Func1()*, care apelează *Func2()*; aceasta va arunca o excepţie. Deoarece lipseşte un event handler care să trateze această excepţie, se va întrerupe thread–ul curent (şi fiind singurul, şi întregul proces) de către CLR, iar la ieşire vom avea:

```
Enter Main...
Enter Func1...
Enter Func2...
Exception occurred: System.Exception: An exception of type
System.Exception was thrown at Test.Func2( )
in ...Test.cs:line 24
at Test.Func1( )
in ...Test.cs:line 18
at Test.Main( )
in ...Test.cs:line 12
```

Deoarece este aruncată o excepţie, în metoda *Func2()* nu se va mai executa ultima linie, ci CLR–ul va începe imediat căutarea event handler–ului care să trateze excepţia. La fel, nu se execută nici ultima linie din *Func1()* sau din *Main()*.

Prinderea cu *catch*

Prinderea şi tratarea excepţiei se poate face folosind un bloc *catch*, creat prin intermediul instrucţiunii *catch*.

Exemplu:

```
using System;
public class Test
{
  public static void Main( )
  {
```

```
    Console.WriteLine(‘‘Enter Main...’’);
    Test t = new Test( );
    t.Func1( );
    Console.WriteLine(‘‘Exit Main...’’);
  }
  public void Func1( )
  {
    Console.WriteLine(‘‘Enter Func1...’’);
    Func2( );
    Console.WriteLine(‘‘Exit Func1...’’);
  }
  public void Func2( )
  {
    Console.WriteLine(‘‘Enter Func2...’’);
    try
    {
      Console.WriteLine(‘‘Entering try block...’’);
      throw new System.Exception( );
      Console.WriteLine(‘‘Exiting try block...’’);
    }
    catch
    {
      Console.WriteLine(‘‘Exception caught and handled.’’);
    }
    Console.WriteLine(‘‘Exit Func2...’’);
  }
}
```

Se observă că s–a folosit un bloc *try* pentru a delimita instrucţiunile care vor duce la apariţia excepţiei. În momentul în care se aruncă excepţia, restul instrucţiunilor din blocul *try* se ignoră şi controlul este preluat de către blocul *catch*. Deoarece excepţia a fost tratată, CLR–ul nu va mai opri procesul. La ieşire se va afişa:

```
Enter Main...
Enter Func1...
Enter Func2...
Entering try block...
Exception caught and handled.
Exit Func2...
Exit Func1...
Exit Main...
```

Se observă că în blocul *catch* nu s–a specificat tipul de excepţie care se prinde; asta înseamnă că se va prinde orice excepţie se va arunca, indiferent de tipul ei. Chiar dacă excepţia este tratată, execuţia nu se va relua de la instrucţiunea care a produs excepţia, ci se continuă cu instrucţiunea de după blocul *catch*.

Uneori, prinderea şi tratatarea excepţiei nu se poate face în funcţia apelată, ci doar în funcţia apelantă. Exemplu:

```
using System;
public class Test
{
  public static void Main( )
  {
    Console.WriteLine(‘‘Enter Main...’’);
    Test t = new Test( );
    t.Func1( );
    Console.WriteLine(‘‘Exit Main...’’);
  }
  public void Func1( )
  {
    Console.WriteLine(‘‘Enter Func1...’’);
    try
    {
      Console.WriteLine(‘‘Entering try block...’’);
      Func2( );
      Console.WriteLine(‘‘Exiting try block...’’);
    }
    catch
    {
      Console.WriteLine(‘‘Exception caught and handled.’’);
    }
    Console.WriteLine(‘‘Exit Func1...’’);
  }
  public void Func2( )
  {
    Console.WriteLine(‘‘Enter Func2...’’);
    throw new System.Exception( );
    Console.WriteLine(‘‘Exit Func2...’’);
  }
}
```

La ieşire se va afişa:

```
Enter Main...
Enter Func1...
Entering try block...
Enter Func2...
Exception caught and handled.
Exit Func1...
Exit Main...
```

Este posibil ca într–o secvenţă de instrucţiuni să se arunce mai multe tipuri de excepţii, în funcţie de natura stării apărute. În acest caz, prinderea excepţiei printr–un bloc *catch* generic, ca mai sus, nu este utilă; am vrea ca în funcţie de natura excepţiei aruncate, să facem o tratare anume. Se sugereaza chiar să nu se folosească această construcţie de prindere generică, deoarece în majoritatea cazurilor este necesar să se cunoască natura erorii (de exemplu pentru a fi scrisă într-un fişier de logging, pentru a fi consultată mai târziu).

Acest lucru se face specificând tipul excepţiei care ar trebui tratate în blocul *catch*:

```
using System;
public class Test
{
  public static void Main( )
  {
    Test t = new Test( );
    t.TestFunc( );
  }
  //incearca sa imparta doua numere
  public void TestFunc( )
  {
    try
    {
      double a = 5;
      double b = 0;
      Console.WriteLine (‘‘{0} / {1} = {2}’’, a, b, DoDivide(a,b));
    }
    //cel mai derivat tip de exceptie se specifica primul
    catch (System.DivideByZeroException)
    {
      Console.WriteLine(‘‘DivideByZeroException caught!’’);
    }
    catch (System.ArithmeticException)
    {
```

```
      Console.WriteLine(‘‘ArithmeticException caught!’’);
    }
    //Tipul mai general de exceptie este ultimul
    catch
    {
      Console.WriteLine(‘‘Unknown exception caught’’);
    }
  }
  //efectueaza impartirea daca se poate
  public double DoDivide(double a, double b)
  {
    if (b == 0)
      throw new System.DivideByZeroException( );
    if (a == 0)
      throw new System.ArithmeticException( );
    return a/b;
  }
}
```

În exemplul de mai sus s–a convenit ca o împărţire cu numitor 0 să ducă la o execepţie *System.DivideByZeroException*, iar o împărţire cu numărător 0 să ducă la apariţia unei excepţii de tip *System.ArithmeticException*. Este posibilă specificarea mai multor blocuri de tratare a excepţiilor. Aceste blocuri sunt parcurse în ordinea în care sunt specificate, iar primul tip care se potriveşte cu excepţia aruncată (în sensul că tipul excepţie specificat este fie exact tipul obiectului aruncat, fie un tip de bază al acestuia - din cauză de upcasting) este cel care va face tratarea excepţiei apărute. Ca atare, este important ca ordinea excepţiilor tratate să fie de la cel mai derivat la cel mai general. În exemplul anterior, *System.DivideByZeroException* este derivat din clasa *System.ArithmeticException*.

Blocul *finally*

Uneori, aruncarea unei excepţii şi golirea stivei până la blocul de tratare a excepţiei poate să nu fie o idee bună. De exemplu, dacă excepţia apare atunci când un fişier este deschis (şi închiderea lui se poate face doar în metoda curentă), atunci ar fi util ca să se închidă fişierul înainte ca să fie preluat controlul de către metoda apelantă. Altfel spus, ar trebui să existe o garanţie că un anumit cod se va executa, indiferent dacă totul merge normal sau apare o excepţie. Acest lucru se face prin intermediul blocului *finally*, care se va executa în orice situaţie. Existenţa acestui bloc elimină necesitatea existenţei blocurilor *catch* (cu toate că şi acestea pot să apară).

Exemplu:

```
using System;
public class Test
{
  public static void Main( )
  {
    Test t = new Test( );
    t.TestFunc( );
  }
  public void TestFunc( )
  {
    try
    {
      Console.WriteLine(‘‘Open file here’’);
      double a = 5;
      double b = 0;
      Console.WriteLine (‘‘{0} / {1} = {2}’’, a, b, DoDivide(a,b));
      Console.WriteLine (‘‘This line may or may not print’’);
    }
    finally
    {
      Console.WriteLine (‘‘Close file here.’’);
    }
  }
  public double DoDivide(double a, double b)
  {
    if (b == 0)
    throw new System.DivideByZeroException( );
    if (a == 0)
    throw new System.ArithmeticException( );
    return a/b;
  }
}
```

În exemplul de mai sus, mesajul “Close file here” se va afişa indiferent de ce parametri se transmit metodei *DoDivide()*.

La aruncarea unei excepţii se poate particulariza obiectul care se aruncă:

```
if (b == 0)
{
  DivideByZeroException e = new DivideByZeroException( );
```

```
  e.HelpLink = ‘‘http://www.greselifatale.com’’;
  throw e;
}
```

iar când excepţia este prinsă, se poate prelucra informaţia:

```
catch (System.DivideByZeroException e)
{
  Console.WriteLine( ‘‘DivideByZeroException!
    goto {0} and read more’’, e.HelpLink);
}
```

Crearea propriilor excepţii

În cazul în care suita de excepţii predefinite nu este suficientă, programatorul îşi poate construi propriile tipuri. Se recomandă ca acestea să fie derivate din *System.ApplicationException*, care este derivată direct din *System.Exception*. Se indică această derivare deoarece astfel se face distincţie între excepţiile aplicaţie şi cele sistem (cele aruncate de către CLR).

Exemplu:

```
using System;
public class MyCustomException : System.ApplicationException
{
  public MyCustomException(string message): base(message)
  {
  }
}

public class Test
{
  public static void Main( )
  {
    Test t = new Test( );
    t.TestFunc( );
  }

  public void TestFunc( )
  {
    try
    {
      double a = 0;
```

```
    double b = 5;
    Console.WriteLine (‘‘{0} / {1} = {2}’’, a, b, DoDivide(a,b));
    Console.WriteLine (‘‘This line may or may not print’’);
  }
  catch (System.DivideByZeroException e)
  {
    Console.WriteLine(‘‘DivideByZeroException! Msg: {0}’’,
      e.Message);
    Console.WriteLine(‘‘HelpLink: {0}’’,
      e.HelpLink);
  }
  catch (MyCustomException e)
  {
    Console.WriteLine(‘‘\nMyCustomException! Msg: {0}’’,
      e.Message);
    Console.WriteLine(‘‘\nHelpLink: {0}\n’’,
      e.HelpLink);
  }
  catch
  {
    Console.WriteLine(‘‘Unknown exception caught’’);
  }
}

public double DoDivide(double a, double b)
{
  if (b == 0)
  {
    DivideByZeroException e = new DivideByZeroException( );
    e.HelpLink= ‘‘http://www.greselifatale.com’’;
    throw e;
  }
  if (a == 0)
  {
    MyCustomException e = new MyCustomException( ‘‘Can’t have
      zero divisor’’);
    e.HelpLink = ‘‘http://www.greselifatale.com/NoZeroDivisor.htm’’;
    throw e;
  }
  return a/b;
}
```

```
}
```

Rearuncarea excepţiilor

Este perfect posibil ca într–un bloc de tratare a excepţiilor să se se facă o tratare primară a excepţiei, după care să se arunce mai departe o altă excepţie, de acelaşi tip sau de tip diferit (sau chiar excepţia originală). Dacă se doreşte ca această excepţie să păstreze cumva în interiorul ei excepţia originală, atunci constructorul permite înglobarea unei referinţe la aceasta; această referinţă va fi accesibilă prin intermediul proprietăţii *InnerException*:

```
using System;
public class MyCustomException : System.ApplicationException
{
  public MyCustomException(string message,Exception inner):
    base(message,inner)
  {
  }
}
public class Test
{
  public static void Main( )
  {
    Test t = new Test( );
    t.TestFunc( );
  }
  public void TestFunc( )
  {
    try
    {
      DangerousFunc1( );
    }
    catch (MyCustomException e)
    {
      Console.WriteLine(``\n{0}'', e.Message);
      Console.WriteLine(``Retrieving exception history...'');
      Exception inner = e.InnerException;
      while (inner != null)
      {
        Console.WriteLine(``{0}'',inner.Message);
        inner = inner.InnerException;
```

```
    }
  }
}
public void DangerousFunc1( )
{
  try
  {
    DangerousFunc2( );
  }
  catch(System.Exception e)
  {
    MyCustomException ex = new MyCustomException(``E3 -
      Custom Exception Situation!'',e);
    throw ex;
  }
}
public void DangerousFunc2( )
{
  try
  {
    DangerousFunc3( );
  }
  catch (System.DivideByZeroException e)
  {
    Exception ex =
    new Exception(``E2 - Func2 caught divide by zero'',e);
    throw ex;
  }
}
public void DangerousFunc3( )
{
  try
  {
    DangerousFunc4( );
  }
  catch (System.ArithmeticException)
  {
    throw;
  }
  catch (System.Exception)
  {
```

```
      Console.WriteLine(‘‘Exception handled here.’’);
    }
  }
  public void DangerousFunc4( )
  {
    throw new DivideByZeroException("E1 - DivideByZero Exception");
  }
}
```

6.3.3 Reîncercarea codului

Se poate pune întrebarea: cum se procedează dacă se dorecşte revenirea la codul care a produs excepţia, după tratarea ei? Există destule situaţii în care reexecutarea acestui cod este dorită: să ne gândim de exemplu la cazul în care într-o fereastră de dialog se specifică numele unuei fişier ce trebuie procesat, numele este introdus greşit si se doreşte ca să se permită corectarea numelui. Un alt exemplu clasic este cazul în care autorul unei metode ştie că o operaţie poate să eşueze periodic – de exemplu din cauza unui timeout pe reţea – dar vrea să reîncerce operaţia de n ori înainte de a semnala eroare.

În această situaţie se poate defini o etichetă înaintea blocului `try` la care să se permită salutl printr–un `goto`. Următorul exemplu permite unui utilizator să specifice de maxim trei ori numele unui fişier ce se procesează, cu revenire în cazul erorii.

```
using System;
using System.IO;

class Retry
{

    static void Main()
    {
        StreamReader sr;

        int attempts = 0;
        int maxAttempts = 3;

        GetFile:
        Console.Write("\n[Attempt #{0}] Specify file " +
            "to open/read: ", attempts+1);
        string fileName = Console.ReadLine();
```

```
        try
        {
            sr = new StreamReader(fileName);

            Console.WriteLine();

            string s;
            while (null != (s = sr.ReadLine()))
            {
                Console.WriteLine(s);
            }
            sr.Close();
        }
        catch(FileNotFoundException e)
        {
            Console.WriteLine(e.Message);
            if (++attempts < maxAttempts)
            {
                Console.Write("Do you want to select " +
                    "another file: ");
                string response = Console.ReadLine();
                response = response.ToUpper();
                if (response == "Y") goto GetFile;
            }
            else
            {
                Console.Write("You have exceeded the maximum " +
                    "retry limit ({0})", maxAttempts);
            }


        }
        catch(Exception e)
        {
            Console.WriteLine(e.Message);
        }

        Console.ReadLine();
}
}
```

6.3.4 Compararea tehnicilor de manipulare a erorilor

Metoda standard de tratare a aerorilor a fost în general returnarea unui cod de eroare către metoda apelantă. Ca atare, apelantul are sarcina de a descifra acest cod de eroare şi să reacţioneze în consecinţă. Însă aşa cum se arată mai jos, tratarea excepţiilor este superioară acestei tehnici din mai multe motive.

Neconsiderarea codului de retur

Apelul unei funcţii care returnează un cod de eroare poate fi făcut şi fără a utiliza efectiv codul returnat, scriind doar numele funcţiei cu parametrii de apel. Dacă de exemplu pentru o anmită procesare se apelează metoda `A` (de exemplu o deschidere de fişier) după care metoda `B` (citirea din fişier), se poate ca în `A` să apară o eroare care nu este luată în considerare; apelul lui `B` este deja sortit eşecului, pentru că buna sa funcţionare depinde de efectele lui `A`. Dacă însă metoda `A` aruncă o excepţie, atunci nici măcar nu se mai ajunge la apel de `B`, deoarece CLR-ul va pasa execuţia unui bloc `catch/finally`. Altfel spus, nu se permite o propagare a erorilor.

Manipularea erorii în contextul adecvat

În cazul în care o metodă A apelează alte metode $B_1, \ldots B_n$, este posibil ca oricare din aceste n metode să cauzeze o eroare (şi să returneze cod adecvat); tratarea erorii din exteriorul lui A este dificilă în acest caz, deoarece ar trebui să se cerceteze toate codurile de eroare posibile pentru a determina motivul apariţiei erorii. Dacă se mai adauga şi apelul de metodă B_{n+1} în interiorul lui A, atunci orice apel al lui A trebuie să includă suplimentar şi verificarea pentru posibilitatea ca B_{n+1} să fi cauzat o eroare. Ca atare, costul menţinerii codului creşte permanent, ceea ce are un impact negativ asupra TCO-ului[5]

Folosind tratarea excepţiilor, aruncând excepţii cu mesaje de eroare explicite sau excepţii de un anumit tip (definit de programator) se poate trata mult mai convenabil o situaţie deosebită. Mai mult decât atât, introducerea apelului lui B_{n+1} în interiorul lui A nu reclamă modificare suplimentară, deoarece tipul de excepţie aruncat de B_{n+1} este deja tratat (desigur, se presupune că se defineşte un tip excepţie sau o ierarhie de excepţii creată convenabil).

Uşurinţa citirii codului

Pentru comparaţie, se poate scrie un cod care realizează procesarea conţinutului unui fişier folosind coduri de eroare returnate sau excepţii. În primul caz,

[5]Total Cost of Ownership.

soluţia va conţine cod de prelucrare a conţinutului mixat cu cod de verificare şi reacţie pentru diferitele cazuri de excepţie. Codul în al doilea caz este mult mai scurt, mai uşor de înţeles, de menţinut, de corectat şi extins.

Aruncarea de excepţii din constructori

Nimic nu opreşte ca o situaţie deosebită să apară într–un apel de constructor. Tehnica verificării codului de retur nu mai funcţionează aici, deoarece un constructor nu returnează valori. Folosirea excepţiilor este în acest caz aproape de neînlocuit.

6.3.5 Sugestie pentru lucrul cu excepţiile

În Java, programatorii trebuie să declare că o metodă poate arunca o excepţie şi să o declare explicti într–o listă astfel încât un apelant să ştie că se poate se poate aştepta la primirea ei. Această cunoaştere în avans permite conceperea unui plan de lucru cu fiecare dintre ele, preferabil decât să se prindă oricare dintre ele cu un *catch* generic. În cazul .NET se sugerează să se menţină o documentaţie cu excepţiile care pot fi aruncate de fiecare metodă.

Curs 7

Colecţii. Clase generice.

7.1 Colecţii

Un vector reprezintă cel mai simplu tip de colecţie. Singura sa deficienţă este faptul că trebuie cunoscut dinainte numărul de elemente conţinute.

Spaţiul de nume *System.Collections* pune la dispoziţie un set de clase de tip colecţie. Clasele din acest spaţiu de nume reprezintă containere de elemente de tip *Object* care îşi gestionează singure necesarul de memorie (cresc pe măsură ce se adaugă elemente; pot de asemenea să îşi reducă efectivul de memorie alocat atunci când numărul de elemente conţinute este prea mic).

Exemplu:

```
ArrayList myCollection = new ArrayList();
myCollection.Add(client);
client = myCollection[0] as Client;
```

Remarcăm conversia explicită pentru recuperarea unui element din colecţie.

Elementele de bază pentru lucrul cu colecţiile sunt un set de interfeţe care oferă o mulţime consistentă de metode de lucru. Principalele colecţii împreună cu interfeţele implementate sunt:

- *ArrayList* : *IList*, *ICollection*, *IEnumerable*, *ICloneable*
- *SortedList* : *IDictionary*, *ICollection*, *IEnumerable*, *ICloneable*
- *Hashtable* : *IDictionary*, *ICollection*, *IEnumerable*, *ISerializable*, *IDeserializationCallback*, *ICloneable*
- *BitArray* : *ICollection*, *IEnumerable*, *ICloneable*

- *Queue* : *ICollection*, *IEnumerable*, *ICloneable*
- *Stack* : *ICollection*, *IEnumerable*, *ICloneable*
- *CollectionBase* : *IList*, *ICollection*, *IEnumerable*
- *DictionaryBase* : *IDictionary*, *ICollection*, *IEnumerable*
- *ReadOnlyCollectionBase* : *ICollection*, *IEnumerable*

IEnumerable

Implementările aceste interfeţe permit iterarea peste o colecţie de elemente. Unica metodă declarată este metoda *GetEnumerator*:

```
IEnumerator GetEnumerator ()
```

unde un obiect de tip *IEnumerator* este folosit pentru parcurgerea colecţiei – un aşa numit iterator.

ICollection

Interfaţa *ICollection* este tipul de bază pentru orice clasă de tip colecţie; se derivează din *IEnumerable* şi prezintă următoarele proprietăţi şi metode:

- *Count* - proprietate de tip întreg care returnează numărul de elemente conţinute în colecţie
- *IsSynchronized* - proprietate logică ce indică dacă colecţia este sincronizată (sigură pentru accesarea de către mai multe fire de execuţie)
- *SyncRoot* - proprietate care returnează un obiect care poate fi folosit pentru a sincroniza accesul la colecţie
- *CopyTo* - metodă care permite copierea conţinutului colecţiei într-un vector

IList

Reprezintă o colecţie de obiecte care pot fi accesate individual printr-un index.

Proprietăţile sunt:

- *IsFixedSize* - returneaza o valoare logică indicând dacă lista are o dimensiune fixă

- *IsReadOnly* - returnează o valoare logică indicând dacă lista poate fi doar citită
- *Item* - returnează sau setează elementul de la locaţia specificată

Metodele sunt:

- *Add* - adaugă un obiect la o listă
- *Clear* - goleşte lista
- *Contains* - determină dacă colecţia conţine o anumită valoare
- *IndexOf* - determină poziţia în listă a unei anumite valori
- *Insert* - inserează o valoare la o anumită poziţie
- *Remove* - şterge prima apariţie a unui obiect din listă
- *RemoveAt* - sterge un obiect aflat la o anumită locaţie

IDictionary

Interfaţa *IDictionary* reprezintă o colecţie de perechi (cheie, valoare). Permite indexarea unei colecţii de elemente după altceva decât indici întregi. Fiecare pereche trebuie să aibă o cheie unică.

Proprietăţile sunt:

- *IsFixedSize*, *IsReadOnly* - returnează o valoare care precizează dacă colecţia este cu dimensiune maximă fixată, respectiv doar citibilă
- *Item* - returnează un element având o cheie specificată
- *Keys* - returnează o colecţie de obiecte conţinând cheile din dicţionar
- *Values* - returnează un obiect de tip colecţie care conţine toate valorile din dicţionar

Metodele sunt:

- *Add* - adaugă o pereche (cheie, valoare) la dicţionar; dacă cheia există deja, se va face suprascrierea valorii asociate
- *Clear* - se şterge conţinutul unui dicţionar
- *Contains* - determină dacă dictionarul conţine un element cu cheia specificată

- *GetEnumerator* - returnează un obiect de tipul *IDictionaryEnumerator* asociat
- *Remove* - şterge elementul din dicţionar având cheia specificată

7.1.1 Iteratori pentru colecţii

Colecţiile (atât cele de tip listă, cât şi cele dictionar) moştenesc interfaţa *IEnumerable* care permite construirea unui obiect de tip enumerator instanţă a lui *IEnumerable*:

```
interface IEnumerator {
  object Current {get;}
  bool MoveNext();
  void Reset();
}
```

Remarcăm că un asemenea iterator este de tip forward-only. Proprietatea *Current* returnează elementul curent al iterării. Metoda *MoveNext* avansează la următorul element al colecţiei, returnând *true* dacă acest lucru s-a putut face şi *false* în caz contrar; trebuie să fie apelată cel puţin o dată înaintea accesării componentelor colecţiei. Metoda *Reset* reiniţializează iteratorul mutând poziţia curentă înaintea primului obiect al colecţiei.

Pentru fiecare clasă de tip colecţie, enumeratorul este implementat ca o clasă internă. Returnarea unui enumerator se face prin apelul metodei *GetEnumerator*.

Exemplu: se va apela în mod explicit metoda de returnare a iteratorului şi mai departe acesta se foloseşte pentru afişarea elementelor.

```
ArrayList list = new ArrayList();
list.Add("One");
list.Add("Two");
list.Add("Three");

IEnumerator e = list.GetEnumerator();
while(e.MoveNext())
{
  Console.WriteLine(e.Current);
}
```

Apelul unui iterator este mecanismul esenţial pentru funcţionara instrucţiunii *foreach*, care debutează prin a apela intern metoda *GetEnumerator* iar trecerea la elementul următor se face cu metoda *MoveNext*. Altfel spus, exemplul de mai sus este echivalent cu:

```
ArrayList list = new ArrayList();
list.Add("One");
list.Add("Two");
list.Add("Three");

foreach(String s in list)
{
  Console.WriteLine(s);
}
```

7.1.2 Colecţii de tip listă

Colecţiile de tip listă sunt: *ArrayList*, *BitArray*, *Stack*, *Queue* şi *CollectionBase*.

ArrayList

Este o clasă concretă care stochează o colecţie de elemente sub forma unui vector auto-redimensionabil. Suportă mai mulţi cititori concurenţi şi poate fi accesat exact ca un vector:

```
ArrayList list = new ArrayList();
list.Add(...);
Console.WriteLine(list[0]);
list[0] = "abc";
```

BitArray

Acest tip de colecţie gestionează un vector de elemente binare reprezentate ca booleeni, unde *true* reprezintă 1 iar *false* 0. Cele mai importante metode sunt:

- *And*, *Or*, *Xor* - produce un nou *BitArray* care conţine rezultatul aplicării operanzilor respectivi pe elementele din colecţia curentă şi altă colecţie dată ca argument. Dacă cele 2 colecţii nu au acelaşi număr de elemente, se aruncă excepţie
- *Not* - returnează un obiect de tip *BitArray* care conţine valorile negate din colecţia curentă

Stack

Stack reprezintă o colecţie ce permite lucrul conform principiului *LIFO - Last In, First Out.*

Queue

Clasa *Queue* este nou apărută în versuinea 2.0 ce permite implementarea politicii *FIFO - First In, First Out*.

CollectionBase

Clasa *CollectionBase* reprezintă o clasă abstractă, bază pentru o colecţie puternic tipizată. Programatorii sunt încurajaţi să deriveze această clasă decât să creeze una proprie de la zero.

7.1.3 Colecţii de tip dicţionar

Colecţiile de tip dicţionar (*SortedList*, *Hashtable*, *DictionaryBase*) conţin obiecte care se manipulează prin intermediul cheii asociate (care poate fi altceva decât un indice numeric). Toate extind interfaţa *IDictionary*, iar ca enumeratorul este de tip *IDictionaryEnumerator*:

```
interface IDictionaryEnumerator : IEnumerator {
  DictionaryEntry Entry {get;}
  object Key {get;}
  object Value {get;}
}
```

unde *DictionaryEntry* este definit ca:

```
struct DictionaryEntry {
  public DictionaryEntry(object key, object value) { ... }
  public object Key {get; set;}
  public object Value {get; set;}
  ...
}
```

Invocarea enumeratorului se poate face fie explicit:

```
Hashtable htable = new Hashtable();
htable.Add("A", "Chapter I");
htable.Add("B", "Chapter II");
htable.Add("App", "Appendix");
IDictionaryEnumerator e = htable.GetEnumerator();
for ( ; e.MoveNext() ; )
  Console.WriteLine(e.Key);
```

fie implicit:

```
foreach (DictionaryEntry s in htable)
  Console.WriteLine(s.Key);
```

Hashtable

Reprezintă o colecţie de perechi de tip (cheie, valoare) care este organizată pe baza codului de dispersie (hashing) al cheii. O cheie nu poate să fie nulă. Obiectele folosite pe post de chei trebuie să suprascrie metodele *Object.GetHashCode* şi *Object.Equals*. Obiectele folosite pe post de cheie trebuie sa fie imuabile (să nu suporte schimbări de stare care să altereze valorile returnate de cele 2 metode spuse anterior).

SortedList

Reprezintă o colecţie de perechi de tip (cheie, valoare) care sunt sortate după cheie şi se pot accesa după cheie sau după index.

DictionaryBase

Reprezintă o clasă de bază abstractă pentru implementarea unui dicţionar utilizator puternic tipizat (valorile să nu fie văzute ca *object*, ci ca tip specificat de programator).

7.2 Crearea unei colecţii

Vom exemplifica în această secţiune modul în care se defineşte o colecţie ce poate fi iterată. Sunt prezentate 2 variante: specifice versiunilor 1.1 şi respectiv 2.0 ale platformei .NET. În ambele cazuri clasa de tip colecţie va implementa intefaţa *IEnumerable*, dar va diferi modul de implementare.

7.2.1 Colecţie iterabilă (stil vechi)

```
using System;
using System.Collections;
class MyCollection : IEnumerable
{
  private int[] continut = {1, 2, 3};
  public IEnumerator GetEnumerator()
  {
    return new MyEnumerator( this );
  }
```

```
  private class MyEnumerator : IEnumerator
  {
    private MyCollection mc;
    private int index = -1;

    public MyEnumerator( MyCollection mc )
    {
      this.mc = mc;
    }

    public object Current
    {
      get
      {
        if (index < 0 || index >= mc.continut.Length)
        {
          return null;
        }
        else return mc.continut[index];
      }
     }

    public bool MoveNext()
    {
      index++;
      return index < mc.continut.Length;
    }

    public void Reset()
    {
      index = -1;
    }
  }
}
```

Remarcăm că clasa imbricată primeşte prin constructor o referinţă la obiectul de tip colecţie, deoarece orice clasă imbricată în C# este automat şi statică, neavând astfel acces la membrii nestatici ai clasei.

Demonstraţia pentru iterarea clasei este:

```
class TestIterator
```

```
{
  static void Main()
  {
    MyCollection col = new MyCollection();
    foreach(int i in col)
    {
      Console.WriteLine(s);
    }
  }
}
```

Instrucţiunea *foreach* va apela iniţial metoda *GetEnumerator* pentru a obţine obiectul de iterare şi apoi pentru acest obiect se va apela metoda *MoveNext* la fiecare iteraţie. Dacă se returnează *true* atunci se apelează automat şi metoda *Current* pentru obţinerea elementului curent din colecţie; dacă se returnează *false* atunci execuţia lui *foreach* se termină.

Implementarea de mai sus permite folosirea simultană a mai multor iteratori, cu păstrarea stării specifice.

Defectele majore ale acestei implementări sunt:

1. Complexitatea codului (numărul mare de linii). Deşi uşor de înţeles şi general acceptată (fiind de fapt un design pattern), abordarea presupune scrierea multor linii de cod, motiv pentru care programatorii evita aceasta facilitate, preferând mecanisme alternative precum indexatorii.

2. Datorită semnăturii metodei *Current* se returnează de fiecare dată un *Object*, pentru care se face fie boxing şi unboxing (dacă in colecţie avem tip valoare - cazul de mai sus), fie downcasting (de la *Object* la tipul declarat in prima parte a lui foreach, dacă in colecţie avem tip referinţă). În primul caz resursele suplimentare de memorie şi ciclii procesor vor afecta negativ performanţa aplicaţiei iar in al doilea caz apare o conversie explicită care dăunează performanţei globale. Modalitatea de evitare a acestei probleme este ca să nu se implementeze interfeţele *IEnumerator* şi *IEnumerable*, ci scriind metoda *Current* astfel incât să returneze direct tipul de date necesar (*int* in cazul nostru). Acest lucru duce insă la expunerea claselor imbricate (văzute ca nişte clase auxiliare), ceea ce incalcă principiul incapsularii. În plus, cantitatea de cod rămane aceeaşi.

Pentru prima problemă vom da varianta de mai jos. Pentru cea de a doua, rezolvarea se dă sub forma claselor generice.

7.2.2 Colecţie iterabilă (stil nou)

Incepand cu C# 2.0 se poate defini un iterator mult mai simplu. Pentru aceasta se foloseste instrucţiunea *yield*. *yield* este folosită într-un bloc de iterare pentru a semnala valoarea ce urmează a fi returnată sau oprirea iterării. Are formele:

```
yield return expression;
yield break;
```

În prima formă se precizează care va fi valoarea returnată; în cea de-a doua se precizează oprirea iterării (sfârşitul secvenţei de elemente de returnat).

Pentru exemplificare, prezentăm o metodă al cărei rezultat este folosit pentru iterare. Valorile returnate de această metodă sunt pătratele numerelor de la 1 la valoarea argumentului

```
using System;
using System.Collections;
using System.Text;

namespace TestCollection
{
    class Program
    {
        static IEnumerable Squares(int number)
        {
            for (int i = 1; i <= number; i++)
            {
                yield return i*i;
            }
        }

        static void Main(string[] args)
        {
            foreach (int iterate in Squares(10))
            {
                Console.WriteLine(iterate.ToString());
            }
        }

    }
}
```

Remarcăm că are loc următorul efect: la fiecare iteraţie se returnează următoarea valoare din colecţie (colecţia este definită de metoda *Squares*). Astfel, se creează impresia că la fiecare iterare din metoda *Main* se reia execuţia din metoda Squares de unde a rămas la apelul precedent; acest mecanism este diferit de cel al rutinelor (metodelor) întâlnite până acum, purtând numele de *corutină*.

Ca să facem evident acest mecanism de continuare a execuţiei de la punctul de retur anterior, considerăm exemplul:

```
using System;
using System.Collections.Generic;

namespace Iterator
{
    class DemoCorutina
    {
        static IEnumerable<int> Numere()
        {
            Console.WriteLine("Returnare 1");
            yield return 1;
            Console.WriteLine("Returnare 2");
            yield return 2;
            Console.WriteLine("Returnare 3");
            yield return 3;
        }

        static void Main(string[] args)
        {
            foreach(int valoare in Numere())
            {
                Console.WriteLine(valoare.ToString());
            }
        }
    }
}
```

pentru care rezultatul afişat pe ecran este:

```
Returnare 1
1
Returnare 2
2
```

```
Returnare 3
3
```

deci în mod clar apelul pentru următoarea valoarea dată de către metoda `Numere` se continuă de la punctul de ieşire anterior. De fapt, compilatorul va genera automat o implementare de metodă de tip *IEnumerable* (precum am făcut manual în secţiunea 7.2.1), permiţându–se astfel programatorului să se concentreze pe designul metodei şi mai puţin pe stufoasele detaliile interne. Un alt aspect demn de reţinut este că secvenţa se construieşte pe măsură ce datele din enumerare sunt parcurse.

Clasa *MyCollection* de mai sus s-ar rescrie astfel:

```
class MyCollection : IEnumerable
{
    private int[] continut = { 1, 2, 3 };
    public IEnumerator GetEnumerator()
    {
        for(int i=0; i<continut.Length; i++)
        {
            yield return continut[i];
        }
    }
}
```

Pentru a demonstra utilitatea acestui tip de implementare, mai jos dăm rezolvarea pentru următoarea problemă: plecându–se de la un arbore binar să se scrie iteratorii pentru parcurgerea în inordine şi preordine. Nu vom prezenta construirea efectivă a arborelui, aceasta fiind o problema separată. Practic, se va implementa în mod recursiv o iterare peste arbore.

Pentru început, definiţia tipului nod:

```
using System;
namespace TestIterTree
{
    class TreeNode<T>
    {
        private T value;
        private TreeNode<T> left, right;

        public T Value
        {
            get
```

```
            {
                return value;
            }
            set
            {
                this.value = value;
            }
        }

        public TreeNode<T> Left
        {
            get
            {
                return left;
            }
            set
            {
                left = value;
            }
        }

        public TreeNode<T> Right
        {
            get
            {
                return right;
            }
            set
            {
                this.right = value;
            }
        }
    }
}
```

Urmează definirea arborelui şi a celor doi iteratori:

```
using System;
using System.Collections.Generic;

namespace TestIterTree
{
```

```
class Tree<T>
{
    private TreeNode<T> root;

    #region popularea arborelui cu valori
    public void AddValues(params T[] value)
    {
        Array.ForEach(value, Add);
    }
    #endregion

    #region tehnici de traversare
    public IEnumerable<T> InOrder()
    {
        return InOrder(root);
    }

    public IEnumerable<T> PreOrder()
    {
        return PreOrder(root);
    }
    #endregion

    #region Private helper methods
    private IEnumerable<T> InOrder(TreeNode<T> node)
    {
        if (node.Left != null)
        {
            foreach (T value in InOrder(node.Left))
            {
                yield return value;
            }
        }
        yield return node.Value;
        if (node.Right != null)
        {
            foreach (T value in InOrder(node.Right))
            {
                yield return value;
            }
        }
```

```
        }

        private IEnumerable<T> PreOrder(TreeNode<T> root)
        {
            yield return root.Value;
            if (root.Left != null)
            {
                foreach (T value in PreOrder(root.Left))
                {
                    yield return value;
                }
            }
            if (root.Right != null)
            {
                foreach (T value in PreOrder(root.Right))
                {
                    yield return value;
                }
            }
        }

        private void Add(T value)
        {
            //Implements adding a value to the tree
        }
        #endregion
    }
}
```

Implementarea de mai sus s–a făcut conform definiţiilor recursive pentru cele două tipuri de parcurgeri (al treilea tip de parcurgere se implementează analog). Invităm cititorul să compare aceste implementări cu cele iterative clasice din teoria structurilor de date. Pe lângă timpul scurt de implementare, se câştigă în claritate şi uşurinţă în exploatare.

7.3 Clase generice

Vom prezenta în cele ce urmează suportul .NET 2.0 pentru clase şi metode generice; acestea sunt blocuri de cod parametrizate care permit scriere unui cod general, ce poate fi ulterior adaptat automat la cerintele specifice ale programatorului.

7.3.1 Metode generice

Să presupunem că dorim să scriem o metodă care să realizeze interschimbarea valorilor a două variabile. Variantele sunt:

1. scrierea unei metode pentru fiecare tip al variabilelor: neelegant, cod mult

2. scrierea unei metode care să folosească un *Object* pe post de variabilă auxiliară; dacă se face apelul pentru 2 variabile de tip şir de caractere, apare eroarea "Cannot convert from 'ref string' to 'ref object"'. În plus, antetul metodei ar permite apel pentru un parametru de tip *string* şi celălalt de tip *int*, ceea ce nu ar trebui să fie admis la compilare.

Singurul mod adecvat de rezolvare a problemei este folosirea unei metode generice, ca mai jos:

```
void Swap<T>(ref T a, ref T b)
{
  T aux;
  aux = a;
  a = b;
  b = aux;
}
```

Apelul acestei metode se face astfel:

```
int x = 3, y=4;
Swap<int>(ref x, ref y);//nu apare boxing/unboxing
string a="a", b="b";
Swap<string>(ref a, ref b);
```

Remarcăm că apelul se face specificând tipul efectiv pentru T. Această specificare poate fi omisă dacă compilatorul poate deduce singur care este tipul efectiv T:

```
bool b1=true, b2=false;
Swap(ref b1, ref b2);
```

Tipul generic T poate fi folosit şi ca tip de retur.

7.3.2 Tipuri generice

Mecanismul de genericitate poate fi extins la clase şi structuri. Dăm mai jos exemplu care modelează noţiunea de punct într–un spaţiu bidimensional. Genericitatea provine din faptul că coordonatele pot fi de tip întreg sau fracţionare.

```
struct Point<T>
{
  private T xPos;
  private T yPos;

  public Point(T xPos, T yPos)
  {
    this.xPos = xPos;
    this.yPos = yPos;
  }

  public T X
  {
    get
    {
    return xPos;
    }
    set
    {
    xPos = value;
    }
  }

  public T Y
  {
    get
    {
    return yPos;
    }
    set
    {
    yPos = value;
    }
  }
```

```
  public override string ToString()
  {
    return string.Format("({0}, {1})", xPos.ToString(),
      yPos.ToString());
  }

  public void Reset()
  {
    xPos = default(T);
    yPos = default(T);
  }
}
```

Utilizarea efectivă ar putea fi:

```
Point<int> p = new Point(10, 10);
Point<double> q = new Point(1.2, 3.4);
```

Observăm că:

- Metodele, deşi cu caracter generic, nu se mai specifică drept generice (nu se mai folosesc simbolurile < şi >), acest lucru fiind implicit
- folosim o supraîncarcare a cuvântului cheie *default* pentru a aduce câmpurile la valorile implicite ale tipului respectiv: 0 pentru numerice, *false* pentru boolean, *null* pentru tipuri referinţă.

Mai adăugăm faptul că o clasă poate avea mai mult de un tip generic drept parametru, exemplele clasice fiind colecţiile generice de tip dicţionar pentru care se specifică tipul cheii şi al valorilor conţinute.

7.3.3 Constrângeri asupra parametrilor de genericitate

Pentru structura de mai sus este posibil să se folosească o instanţiere de tipul:

```
Point<StringBuilder> r = null;
```

ceea ce este aberant din punct de vedere semantic. Am dori să putem face restriţionarea tipului parametrilor generici. Un asemenea mecanism există şi permite 5 tipuri de restricţii:

where T:struct	T trebuie să fie tip derivat din *System.ValueType*
where T:class	T trebuie să nu fie derivat din *System.ValueType*
where T:new()	T trebuie să aibe un constructor implicit (fără parametri)
where T:NameOfBaseClass	T trebuie să fie derivat (direct sau nu) din *NameOfBaseClass*
where T:NameOfInterface	T trebuie să implementeze interfaţa *NameOfInterface*

Exemple:

- *class MyGenericClass<T> where T:new()* specifică faptul că parametrul T trebuie să fie un tip cu constructor implicit
- *class MyGenericClass<T> where T:class, IDrawable, new()* specifică faptul că parametrul T trebuie să fie de tip referinţă, să implementeze *IDrawable* şi să posede constructor implicit
- *class MyGenericClass<T>:MyBase, ICloneable where T:struct* descrie o clasă care este derivată din *MyBase*, implementează *ICloneable* iar parametrul T este de tip valoare (structură sau enumerare).

Clasele generice pot fi de asemenea clase de bază pentru tipuri (generice sau nu):

```
class MyList<T>...
class MyStringList : MyList<String>...
```

7.3.4 Interfeţe şi delegaţi generici

Interfeţele şi delegaţii pot fi declarate ca fiind generice; deşi nu pezintă cerinţe sau particularităţi faţă de ceea ce s-a spus mai sus, le evidenţiem astfel doarece gradul înalt de abstractizare le face deosebit de utile în modelarea unui sistem soft complex.

```
interface IMyFeature<T>
{
T MyService(T param1, T param2);
}
```

respectiv:

```
delegate void MyGenericDelegate<T>(T arg);
```

7.4 Colecţii generice

7.4.1 Probleme cu colecţiile de obiecte

Colecţiile, aşa cum au fost ele prezentate în secţiunea 7.1 sunt utile, dar au câteva puncte slabe.

1. să presupunem că pornim cu o listă de tip *ArrayList* la care adăugăm elemente de tip întreg:

   ```
   ArrayList al = new ArrayList();
   al.Add(1);
   al.Add(2);
   int x = (int)al[0];
   ```

   Secvenţa este corectă din punct de vedere sintactic, dar la rulare solicită folosirea mecanismului de boxing şi unboxing. Deşi pentru colecţii mici acest lucru nu are are efecte sesizabile, pentru un număr mare de adăugări sau accesări ale elementelor din listă avem un impact negativ ce trebuie luat în calcul. Am prefera ca tipurile colecţie să suporte lucrul cu tipuri valoare fără costul suplimentar introdus de boxing/unboxing.

2. problema tipului efectiv stocat în colecţie: să presupunem că într–o listă adăugăm:

   ```
   al.Add(new Dog("Miki"));
   al.Add(new Dog("Gogu"));
   al.Add(new Matrix(3, 5));
   Dog dog = (Dog)al[2];
   ```

   Secvenţa de sus este corectă din punct de vedere sintactic, dar la rulare va determina aruncarea unei excepţii de tipul *InvalidCastException*. E de dorit ca la compilare să se poată semnala această greşeală.

7.4.2 Colecţii generice

Clasele generice împreună cu colecţiile au fost combinate în biblioteca .NET Framework, ducând la apariţia unui nou spaţiu de nume, conţinut în *System.Collections*: *System.Collections.Generic*. Acesta conţine tipurile: *ICollection<T>*, *IComparer<T>*, *IDictionary<K, V>*, *IEnumerable<T>*, *IEnumerator<T>*, *IList<T>*, *Queue<T>*, *Stack<T>*, *LinkedList<T>*, *List<T>*.

Exemplu de utilizare:

```
List<int> myInts = new List<int>();
myInts.Add(1);
myInts.Add(2);
//myInts.Add(new Complex());//eroare de compilare
```

Deşi în secvenţa de mai sus tipul listei este *int*, nu se apelează la boxing/unboxing, deoarece lista este compusă din elemente de tip întreg şi nu din obiecte de tip *Object*.

7.5 Elemente specifice C# 3.0

Secţiunea conţine o prezentare a elementelor noi introduse de versiunea 3.0 a limbajului C#, aspecte ce se pot raporta la ceea ce s-a prezentat până acum. Prezentarea este completată în capitolele ulterioare (LINQ, lambda expresii).

7.5.1 Proprietăţi implementate automat

Considerăm clasa:

```
class MyClass
{
  private int myField;

  public int MyField
  {
    get
    {
      return myField;
    }
    set
    {
      myField = value;
    }
  }
}
```

Deseori se pune problema scrierii unor câmpuri private, pentru care accesarea se face prin intermediul proprietăţilor. Este contraindicat să se expună câmpurile ca fiind publice, deoarece se sparge încapsularea şi nu se poate face databinding la câmpuri publice. Dacă un câmp se expune ca fiind public şi

un cod client începe să îl acceseze, este imposibil ca ulterior să se impună cod de validare pentru accesul la el.

Deoarece codul de tipul celui scris mai sus apare foarte des, s-a pus problema simplificării lui. În C# 3.0 se scrie echivalent:

```
class MyClass
{
  public int MyField
  {
    get;
    set;
  }
}
```

Se foloseşte aici mecanismul de implementare automată a unei proprietăţi care acţionează asupra unor câmpuri private autodeclarate; această proprietate returnează sau accesează direct câmpul asociat.

Particularităctile sunt următoarele:

1. nu se declară câmpul privat; acesta este creat automat de compilator, pe baza proprietăţii auto-implementate;

2. nu se scriu implementări pentru `get` şi `set`; corpul lor este caracterul ";". `get` accesează câmpul autodeclarat, `set` setează valoarea câmpului cu ce se află în dreapta semnului egal;

3. nu se poate accesa câmpul autodeclarat altfel decât prin intermediul proprietăţii

4. proprietatea nu poate fi doar read-only sau write-only, ci read-write.

Dacă ulterior se decide implementarea unui accesor, atunci şi celălalt trebuie implementat şi câmpul privat trebuie declarat. Important este însă că se scrie un minim de cod pentru a genera un contract: câmp privat accesat prin proprietate publică.

7.5.2 Iniţializatori de obiecte

Să considerăm clasa:

```
class Person
{
    private string firstName;
```

```
    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }
    private string lastName;
    public string LastName
    {
        get { return lastName; }
        set { lastName = value; }
    }
    private int age;
    public int Age
    {
        get { return age; }
        set { age = value; }
    }
}
```

Se poate scrie următoarea secvenţă de cod care iniţializează o persoană cu datele cuvenite:

```
Person p = new Person();
p.FirstName = "Rafael";
p.Age = 25;
p.LastName = "Popescu";
```

În C# 3.0 se poate scrie mai succint:

```
Person p = new Person { FirstName = "Rafael",
    Age = 25, LastName = "Popescu" };
```

cu acelaşi rezultat[1],[2]. Într-un astfel de caz se face mai întâi apelarea constructorului implicit (indiferent de cine anume îl scrie - compilatorul sau programatorul) şi apoi se face accesarea proprietăţilor, în ordinea scrisă la iniţializator. Membrii pentru care se face iniţializare trebuie să fie publici; în particular, ei

[1]În unele lucrări se foloseşte: `Person p = new Person(){FirstName="Rafael", Age=25, LastName="Popescu" };` deci cu paranteze rotunde după numele clasei folosite de operatorul `new`.

[2]Cele două secvenţe nu sunt totuşi echivalente, aşa cum se arată în `http://community.bartdesmet.net/blogs/bart/archive/2007/11/22/c-3-0-object-initializers-revisited.aspx`

pot fi şi câmpuri, dar acest lucru nu este încurajat de principiul încapsulării. Putem avea inclusiv proprietăţi auto-implementate (secţiunea 7.5.1).

Dacă se scrie un constructor care preia un parametru dar nu şi unul care să fie implicit, de exemplu:

```
public Person(String firstName)
{
    FirstName = firstName;
}
```

atunci se poate încă folosi mecansimul de iniţializare:

```
Person r = new Person("Rafael") {LastName="Popescu", Age = 25 };
```

Exemplul se poate dezvolta mai departe prin exemplificarea construirii unor obiecte mai complexe:

```
Person p = new Person{
  FirstName = "Rafael",
  LastName = "Popescu",
  Age={25},
  Address = new Address{
    City = "Brasov",
    Country = "Romania",
    Street = "Iuliu Maniu"
  }
}
```

7.5.3 Iniţializatori de colecţii

Se dă secvenţa de cod:

```
List<String> list = new List<String>();
list.Add("a");
list.Add("b");
list.Add("c");
```

În C# 3.0 ea este echivalentă cu:

```
List<String> list = new List<String>(){"a", "b", "c"};
```

ceea ce aduce aminte de o trăsătură similară de la iniţializarea tablourilor; mai exact codul de mai jos:

```
String[] x = new String[3];
x[0] = "a";
x[1] = "b";
x[2] = "c";
```

este încă din prima versiune de C# echivalentă cu:

```
String[] x = new String[]{"a", "b", "c"};
```

Iniţializarea colecţiilor vine să ofere acelaşi mecanism ca şi în cazul tablourilor.

Exemplul poate fi completat cu popularea unei colecţii de obiecte compuse:

```
List<Person> persons = new List<Person>(){
    new Person{FirstName = "Rafael", LastName="Popescu", Age=25},
    new Person{FirstName = "Ioana", LastName="Ionescu", Age=23}
};
```

Curs 8

ADO.NET

8.1 Ce reprezintă ADO.NET?

ADO.NET reprezintă o parte componentă a lui .NET Framework ce permite aducerea, manipularea şi modificarea datelor. În mod normal, o sursă de date poate să fie o bază de date, dar de asemenea un fişier text sau Excel sau XML sau Access. Lucrul se poate face fie conectat, fie deconectat de la sursa de date. Deşi există variate modalităţi de lucru cu bazele de date, ADO.NET se impune tocmai prin faptul că permite lucrul deconectat de la baza de date, integrarea cu XML, reprezentarea comună a datelor cu posibilitatea de a combina date din variate surse, toate pe baza unor clase .NET.

Faptul că se permite lucrul deconectat de la sursa de date rezolvă următoarele probleme:

- menţinerea conexiunilor la baza de date este o operaţie costisitoare. O bună parte a lăţimii de bandă este menţinută ocupată pentru nişte procesări care nu necesită neapărat conectare continuă
- probleme legate de scalabilitatea aplicaţiei: se poate ca serverul de baze de date să lucreze uşor cu 5-10 conexiuni menţinute, dar dacă numărul acestora creşte aplicaţia poate să reacţioneze extrem de lent
- pentru unele servere se impun clauze asupra numărului de conexiuni ce se pot folosi simultan.

Toate acestea fac ca ADO.NET să fie o tehnologie mai potrivită pentru dezvoltarea aplicaţiilor Internet decât cele precedente (e.g. ADO, ODBC).

Pentru o prezentare a metodelor de lucru cu surse de date sub platformă Windows se poate consulta [7].

Vom exemplifica în cele ce urmează preponderent pe bază de date Microsoft SQL 2005 Express Edition, ce se poate descărca gratuit de pe site-ul Microsoft.

8.2 Furnizori de date în ADO.NET

Din cauza existenţei mai multor tipuri de surse de date (de exemplu, a mai multor producători de servere de baze de date) e nevoie ca pentru fiecare tip major de protocol de comunicare să se folosească o bibliotecă specializată de clase. Toate aceste clase implementează nişte interfeţe bine stabilite, ca atare trecerea de la un SGBD la altul se face cu eforturi minore (dacă codul este scris ţinând cont de principiile programării orientate pe obiecte).

Există următorii furnizori de date[1] (lista nu este completă):

Tabelul 8.1: Furnizori de date.

Nume furnizor	Prefix API	Descriere
ODBC Data Provider	Odbc	Surse de date cu interfaţă ODBC (baze de date vechi)
OleDb Data Provider	OleDb	Surse de date care expun o interfaţă OleDb, de exemplu Access şi Excel sau SQL Sever versiune mai veche de 7.0
Oracle Data Provider	Oracle	SGBD Oracle
SQL Data Provider	Sql	Pentru interacţiune cu Microsoft SQL Server 7.0, 2000, 2005
Borland Data Provider	Bdp	Acces generic la SGBD-uri precum Interbase, SQL Server, IBM DB2, Oracle
MySql	MySql	SGBD MySql

Prefixele trecute în coloana "Prefix" sunt folosite pentru clasele de lucru specifice unui anumit furnizor ADO.NET: de exemplu, pentru o connexiune SQL Server se va folosi clasa SqlConnection.

8.3 Componentele unui furnizor de date

Fiecare furnizor de date ADO.NET constă în patru componente: *Connection*, *Command*, *DataReader*, *DataAdapter*. Arhitectura ADO.NET este prezentată în figura 8.1

[1] Engl: Data Providers

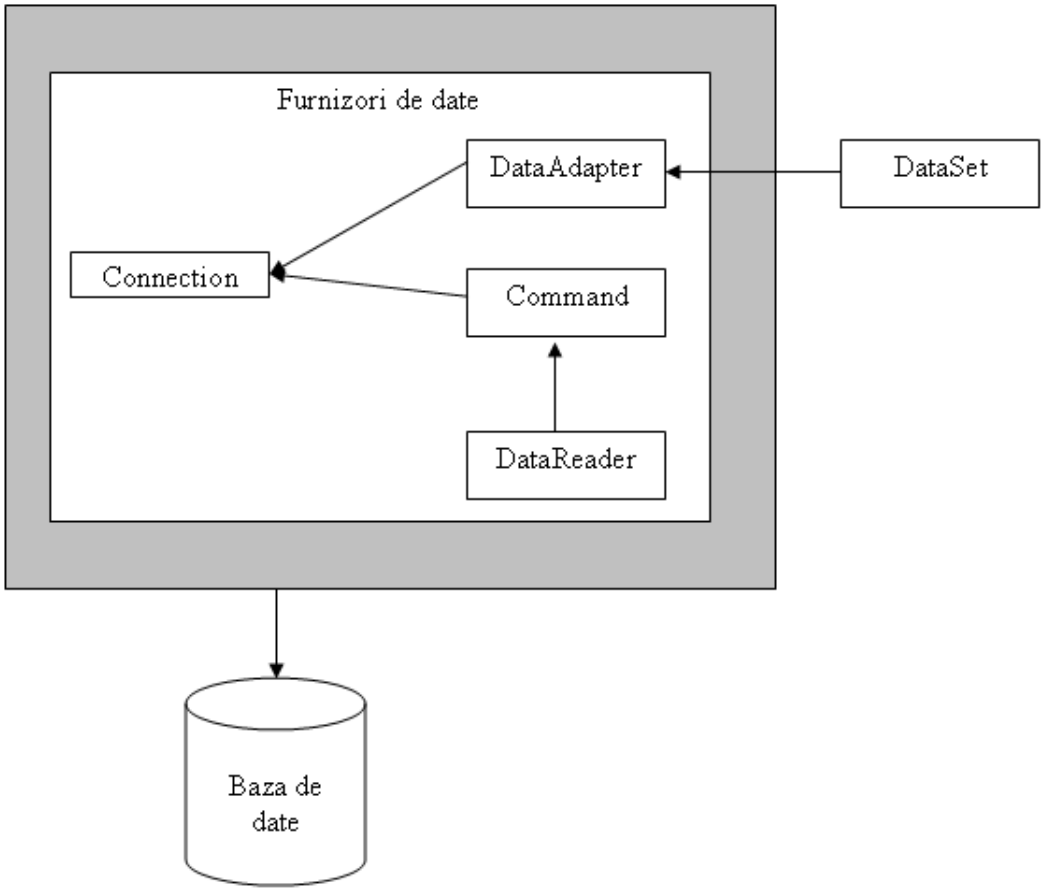

Figura 8.1: Principalele clase ADO.NET

Mai jos sunt date descrieri succinte ale claselor cel mai des utilizate.

8.3.1 Clasele *Connection*

Sunt folosite pentru a reprezenta o conexiune la surse de date. Ele conţin date specifice conexiunii, cum ar fi locaţia sursei de date, numele şi parola contului de acces, etc. În plus au metode pentru deschiderea şi închiderea conexiunilor, pornirea unei tranzacţii sau setarea perioadei de time-out. Stau la baza oricărei accesări de servicii de pe server.

8.3.2 Clasele *Command*

Sunt folosite pentru a executa diferite comenzi pe baza de date (SELECT, INSERT, UPDATE, DELETE) şi pentru a furniza un obiect de tip *DataReader*

sau pentru a umple un *DataSet* prin intermediul unui obiect *DataAdapter*. Pot fi folosite pentru apelarea de proceduri stocate aflate pe server. Ele permit scrierea de interogări SQL parametrizate sau specificarea parametrilor pentru procedurile stocate.

8.3.3 Clasele *DataReader*

Permit navigarea de tip forward–only, read–only în mod conectat la sursa de date. Se obţin pe baza unui obiect de tip *Command* prin apelul metodei *ExecuteReader()*. Accesul rezultat este extrem de rapid cu minim de resurse consumate.

8.3.4 Clasele *DataAdapter*

Ultima componentă principală a unui furnizor de date .NET este *DataAdapter*. Funcţionează ca o punte între sursa de date şi obiecte de tip *DataSet* deconectate, permiţând efectuarea operaţiilor pe baza de date. Conţin referinţe către obiecte de tip *Connection* şi deschid / închid singure conexiunea la baza de date. În plus, un *DataAdapter* conţine referinţe către patru comenzi pentru selectare, ştergere, modificare şi adăugare la baza de date.

8.3.5 Clasa *DataSet*

Această clasă nu este parte a unui furnizor de date .NET ci independentă de particularităţile de conectare şi lucru cu o bază de date anume. Prezintă marele avantaj că poate sa lucreze deconectat de la sursa de date, facilitând stocarea şi modificarea datelor local, apoi reflectarea acestor modificări în baza de date. Un obiect *DataSet* este de fapt un container de tabele şi relaţii între tabele. Foloseşte serviciile unui obiect de tip *DataAdapter* pentru a-şi procura datele şi a trimite modificările înapoi către baza de date. Datele sunt stocate de un *DataSet* în format XML; acelaşi format este folosit pentru transportul datelor.

8.4 Obiecte *Connection*

Clasele de tip *Connection* pun la dispoziţie tot ceea ce e necesar pentru conectarea la baze de date. Este primul obiect cu care un programator ia contact atunci când încearcă să folosească un furnizor de date .NET. Înainte ca o comandă să fie executată pe o bază de date trebuie stabilite datele de conectare şi deschisă conexiunea.

Orice clasă de tip conexiune (din orice furnizor de date) implementează intefaţa *IDbConnection*. De exemplu, clasele *SqlConnection* (folosită pentru conectare la server Microsoft SQL Server 2000) sau *OleDbConnection* (folosită pentru conectare la fişiere .mdb din Access sau Excel) implementează *IDbConnection*.

Pentru deschiderea unei conexiuni se poate proceda ca mai jos:

```
using System.Data.SqlClient;//spatiul de nume SqlClient
...
SqlConnection cn = new SqlConnection(@"Data Source=
localhost\sqlexpress;Database=Northwind;User ID=sa;
Password=parola");
cn.Open();
...
```

Mai sus s–a specificat numele calculatorului pe care se află instalat severul SQL (”localhost”) precum şi al numelui de instanţă pentru acest server (“express”), baza de date la care se face conectarea (“Northwind”), contul SQL cu care se face accesul (“sa”) şi parola pentru acest cont (“parola”).

Pentru conectarea la un fişier Access *Northwind.mdb* aflat în directorul c:\lucru se foloseşte un obiect de tipul OleDbConnection sub forma:

```
using System.Data.OleDb;//spatiul de nume OleDb
...
OleDbConnection cn = new OleDbConnection(
@"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=
C:\Lucru\Northwind.mdb");
cn.Open();
...
```

S-a specificat furnizorul de date (Microsoft.Jet.OLEDB.4.0 pentru fişier Access) precum şi locul unde se află sursa de date (C:\Lucru\Northwind.mdb).

Vom enumera principalele proprietăţi, metode şi evenimente pentru un obiect de tip *Connection*.

8.4.1 Proprietăţi

1. *ConnectionString*: de tip *String*, cu accesori de *get* şi *set*; această proprietate defineşte un string care permite identificarea tipului şi locaţiei sursei de date la care se face conectarea şi eventual contul şi parola de acces. Acest string conţine lista de parametri necesari pentru conectare sub forma numeParametru=valoare, separaţi prin punct şi virgulă. Parametrii sunt:

- *provider*: se specifică furnizorul de date pentru conectarea la sursa de date. Acest furnizor trebuie precizat doar dacă se foloseşte OLE DB .NET Data Provider, însă nu se specifică pentru conectare la SQL Server.
- *Data Source* (sinonim cu *server*): se specifică numele serverului de baze de date sau numele fişierului de date.
- *Initial Catalog* (sinonim cu *Database*): specifică numele baze de date. Baza de date trebuie să se găsească pe serverul dat în *Data Source*.
- *User ID* (sinonim cu *uid*): specifica un nume de utilizator care are acces de loginare la server.
- *Password* (sinonim cu *pwd*): specifică parola contului de mai sus.

2. *ConnectionTimeout*: de tip int, cu accesor de *get*, valoare implicită 15; specifică numărul de secunde pentru care un obiect de conexiune ar trebui să aştepte pentru realizarea conectării la server înainte de a se genera o excepţie. Se poate specifica o valoare diferită de 15 în *ConnectionString* folosind parametrul *Connect Timeout*:

```
SqlConnection cn = new SqlConnection("Data Source=serverBD;
Database=Northwind;User ID=sa;Password=parola;
Connect Timeout=30");
```

   Se poate specifica pentru *Connect Timeout* valoarea 0 cu semnificaţia "aşteaptă oricât", dar se sugerează să nu se procedeze în acest mod.

3. *Database*: atribut de tip string, read-only, returnează numele bazei de date la care s–a făcut conectarea. Folosită pentru a arăta unui utilizator care este baza de date pe care se face operarea.

4. *Provider*: atribut de tip string, read-only, returnează numele furnizorului OLE DB.

5. *ServerVersion*: atribut de tip string, read-only, returnează versiunea de server la care s–a făcut conectarea.

6. *State*: atribut de tip enumerare ConnectionState , read-only, returnează starea curentă a conexiunii. Valorile posibile sunt: *Broken*, *Closed*, *Connecting*, *Executing*, *Fetching*, *Open*.

8.4.2 Metode

1. *Open()*: deschide o conexiune la baza de date
2. *Close(), Dispose()*: închid conexiunea şi eliberează toate resursele alocate pentru ea
3. *BeginTransaction()*: pentru executarea unei tranzacţii pe baza de date; la sfârşit se apelează *Commit()* sau *Rollback()*.
4. *ChangeDatabase()*: se modifică baza de date la care se vor face conexiunile. Noua bază de date trebuie să existe pe acelaşi server ca precedenta.
5. *CreateCommand()*: creează un obiect de tip *Command* valid (care implementează interfaţa *IDbCommand*) asociat cu conexiunea curentă.

8.4.3 Evenimente

Un obiect de tip conexiune poate semnala două evenimente:

- evenimentul *StateChange*: apare atunci când se schimbă starea conexiunii. Event-handlerul este de tipul delegat *StateChangeEventHandler*, care spune care sunt stările între care s–a făcut tranziţia.
- evenimentul *InfoMessage*: apare atunci când furnizorul trimite un avertisment sau un mesaj informaţional către client.

8.4.4 Stocarea stringului de conexiune în fişier de configurare

În general este contraindicat ca stringul de conexiune să fie scris direct în cod; modificarea datelor de conectare (de exemplu parola pe cont sau sursa de date) ar însemna recompilarea codului.

.NET Framework permite menţinerea unor perechi de tipul chei-valoare, specifice aplicaţiei; fişierul este de tip XML. Pentru aplicaţiile Web fişierul se numeşte *web.config*, pentru aplicaţiile de tip consolă fişierul de configurare are extensia *config* şi numele aplicaţiei, iar pentru aplicaţiile Windows acest fişier nu există implicit, dar se adaugă: Project->Add new item->Application Configuration File, implicit acesta având numele App.config. Elementul rădăcină împreună cu declaraţia de XML sunt:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
</configuration>
```

În interiorul elementului rădăcină *configuration* se va introduce elementul *appSettings*, care va conţine oricâte perechi cheie-valoare în interiorul unui atribut numit *add*, precum mai jos:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="constring"
      value="Data Source=localhost\sqlexpress;database=Northwind;
      User ID=sa;pwd=parola"/>
  </appSettings>
</configuration>
```

Clasele neceasre pentru accesarea fişierului de configurare se găsesc în spaţiul de nume *System.Configuration*. Pentru a se putea folosi acest spaţiu de nume trebuie să se adauge o referinţă la assembly-ul care conţine deaceastă clasă: din Solutin explorer click dreapta pe proiect->Add reference...->se alege tab-ul .NET şi de acolo System.Configuration. Utilizarea stringului de conexiune definit anterior se face astfel:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Configuration;
public class UsingConfigSettings
{
  public static void Main()
  {
    SqlConnection con = new SqlConnection(
    ConfigurationManager.AppSettings["constring"];
    //se lucreaza cu conexiunea
    con.Close();
  }
}
```

Este posibil ca într-o aplicaţie să se folosească mai multe conexiuni, motiv pentru care se sugerează ca în loc de varianta precedentă să se folosească elementul XML `<connectionStrings>`:

```
<configuration>
  <appSettings>...</appSettings>
  <connectionStrings>
     <add name ="SqlProviderPubs" connectionString =
```

```
      "Data Source=localhost\sqlexpress;uid=sa;pwd=;
      Initial Catalog=Pubs"/>
      <add name ="OleDbProviderPubs" connectionString =
      "Provider=SQLOLEDB.1;Data Source=localhost;uid=sa;pwd=;
      Initial Catalog=Pubs"/>
  </connectionStrings>
</configuration>
```

Preluarea unui string de conexiune se face prin:

```
string cnStr =
  ConfigurationManager.ConnectionStrings["SqlProviderPubs"]
    .ConnectionString;
```

8.4.5 Gruparea conexiunilor

Gruparea conexiunilor[2] reprezintă reutilizarea resurselor de tip conexiune la o bază de date. Atunci când se creează o grupare de conexiuni se generează automat mai multe obiecte de tip conexiune, acest număr fiind egal cu minimul setat pentru gruparea respectivă. O nouă conexiune este creată dacă toate conexiunile sunt ocupate şi se cere conectare. Dacă dimensiunea maximă setată a grupării este atinsă, atunci nu se va mai crea o conexiune nouă, ci se va pune într–o coadă de aşteptare. Dacă aşteparea durează mai mult decât este precizat în valoarea de Timeout se va arunca o excepţie. Pentru a returna o conexiune la grupare trebuie apelată metoda *Close()* sau *Dispose()* pentru acea conexiune.

Sursele de date .NET administrează automat gruparea de conexiuni, degrevându-l pe programator de aceast aspect ce nu ţine de logica aplicaţiei. La dorinţă comportamentul implicit se poate modifica prin intemediul conţinutului stringului de conectare.

8.4.6 Mod de lucru cu conexiunile

Se cere ca o conexiune să fie întotdeauna închisă (şi dacă se poate, cât mai repede posibil). Ca atare, este de preferat ca să se aplice o schemă de lucru de tipul:

```
IDBConnection con = ...
try
{
```

[2]Engl: connection pooling

```
  //deschidere conexiune
  //lucru pe baza
}
catch(Exception e)
{
  //tratare de exceptie
}
finally
{
  con.Close();
}
```

Deoarece se garantează ca blocul finally este executat indiferent dacă apare sau nu o excepţie, în cazul de mai sus se va închide în mod garantat conexiunea. În acelaşi scop se mai poate folosi şi instrucţiunea `using` (secţiunea 3.5.8).

8.5 Obiecte *Command*

Un clasă de tip *Command* dată de un furnizor .NET trebuie să implementeze interfaţa *IDbCommand*, ca atare toate vor asigura un set de servicii bine specificat. Un asemenea obiect este folosit pentru a executa comenzi pe baza de date: SELECT, INSERT, DELETE, UPDATE sau apel de proceduri stocate (dacă SGBD-ul respectiv ştie acest lucru). Comanda se poate executa numai dacă s-a deschis o conexiune la baza de date.

Exemplu:

```
SqlConnection con = new SqlConnection(
  ConfigurationManager.ConnectionStrings["constring"]
    .ConnectionString;
SqlCommand cmd = new SqlCommand("SELECT * FROM Customers", con);
```

Codul care utilizează o comandă pentru lucrul cu fisiere mdb sau xls ar fi foarte asemănător, cu diferenţa că în loc de *SqlCommand* se foloseşte *OleDbCommand*, din spaţiul de nume *System.Data.OleDb*. Nu trebuie modificat altceva, doarece locaţia sursei de date se specifică doar la conexiune.

Se observă că obiectul de conexiune este furnizat comenzii create. Enumerăm mai jos principalele proprietăţi şi metode ale unui obiect de tip comandă.

8.5.1 Proprietăţi

1. *CommandText*: de tip *String*, cu ambii accesori; conţine comanda SQL sau numele procedurii stocate care se execută pe sursa de date.

2. *CommandTimeout*: de tip int, cu ambii accesori; reprezintă numărul de secunde care trebuie să fie aşteptat pentru executarea interogării. Dacă se depăşeste acest timp, atunci se aruncă o excepţie.

3. *CommandType*: de tip *CommandType* (enumerare), cu ambii accesori; reprezintă tipul de comandă care se execută pe sursa de date. Valorile pot fi:

   - *CommandType.StoredProcedure* - interpretează comanda conţinută în proprietatea *CommandText* ca o un apel de procedură stocată definită în baza de date.
   - *CommandType.Text* - interpretează comanda ca fiind o comandă SQL clasică; este valoarea implicită.
   - *CommandType.TableDirect* - momentan disponibil numai pentru furnizorul OLE DB (deci pentru obiect de tip OleDbCommand); dacă proprietatea *CommandType* are această valoare, atunci proprietatea *CommandText* este interpretată ca numele unui tabel pentru care se aduc toate liniile şi coloanele la momentul executării.

4. *Connection* - proprietate de tip *System. Data. [.NET Data Provider]. PrefixConnection*, cu ambii accesori; conţine obiectul de tip conexiune folosit pentru legarea la sursa de date; "Prefix" este prefixul asociat furnizorului respectiv (a se vedea tabelul 8.1).

5. *Parameters* - proprietate de tip *System.Data.[.NET Data Provider]. PrefixParameterCollection*, read-only; returnează o colecţie de parametri care s-au transmis comenzii; această listă a fost creată prin apelul metodei *CreateParameter()*."Prefix" reprezintă acelasi lucru ca mai sus.

6. *Transaction* - proprietate de tip *System.Data.[.NET Data Provider]. PrefixTransaction*, read-write; permite accesul la obiectul de tip tranzacţie care se cere a fi executat pe sursa de date.

8.5.2 Metode

1. *Constructori* - un obiect de tip comandă poate fi creat şi prin intermediul apelului de constructor; de exemplu un obiect *SqlCommand* se poate obţine astfel:

```
SqlCommand cmd;
cmd = new SqlCommand();
cmd = new SqlCommand(string CommandText);
```

```
cmd = new SqlCommand(string CommandText, SqlConnection con );
cmd = new SqlCommand(string CommandText, SqlConnection con,
     SqlTransaction trans);
```

2. *Cancel()* - încearcă să oprească o comandă dacă ea se află în execuţie. Dacă nu se află în execuţie, sau această încercare nu are nici un efect nu se intâmplă nimic.

3. *Dispose()* - distruge obiectul comandă.

4. *ExecuteNonQuery()* - execută o comandă care nu returnează un set de date din baza de date; dacă comanda a fost de tip INSERT, UPDATE, DELETE, se returnează numărul de înregistrări afectate. Dacă nu este definită conexiunea la baza de date sau aveasta nu este deschisă, se aruncă o excepţie de tip *InvalidOperationException*.

   Exemplu:

   ```
   SqlConnection con = new SqlConnection(
      ConfigurationManager.ConnectionStrings["constring"]
        .ConnectionString);
   SqlCommand cmd = new SqlCommand();
   cmd.CommandText = "DELETE FROM Customers WHERE CustomerID = 'SEVEN'";
   cmd.Connection = con;
   con.Open();
   Console.WriteLine(cmd.ExecuteNonQuery().ToString());
   con.Close()
   ```

   În exemplul de mai sus se returnează numărul de înregistrări care au fost şterse.

5. *ExecuteReader()* - execută comanda conţinută în proprietatea *CommandText* şi se returnează un obiect de tip *IDataReader* (e.g. *SqlDataReader* sau *OleDbDataReader*).

   Exemplu: se obţine conţinutul tabelei Customers într–un obiect de tip *SqlDataReader* (se presupune că baza de date se stochează pe un server MSSQL):

   ```
   SqlConnection con = new SqlConnection(
     ConfigurationManager.ConnectionStrings["constring"]
       .ConnectionString);
   SqlCommand cmd = new SqlCommand();
   ```

```
cmd.CommandText = "SELECT * FROM Customers";
cmd.Connection = con;
con.Open();
SqlDataReader reader = cmd.ExecuteReader();
while(reader.Read())
{
  Console.WriteLine("{0} - {1}",
  reader.GetString(0),
  reader.GetString(1));
}
reader.Close();
con.Close();
```

Metoda *ExecuteReader()* mai poate lua un argument opţional de tip enumerare *CommandBehavior* care descrie rezultatele şi efectul asupra bazei de date:

- *CommandBehavior.CloseConnection* - conexiunea este închisă atunci când obiectul de tip *IDataReader* este închis.
- *CommandBehavior.KeyInfo* - comanda returnează metadate despre coloane şi cheia primară.
- *CommandBehavior.SchemaOnly* - comanda returnează doar informaţie despre coloane.
- *CommandBehavior.SequentialAccess* - dă posibilitatea unui *DataReader* să manipuleze înregistrări care conţin câmpuri cu valori binare de mare întindere. Acest mod permite încărcarea sub forma unui flux de date folosind *GetChars()* sau *GetBytes()*.
- *CommandBehavior.SingleResult* - se returnează un singur set de rezultate
- *CommandBehavior.SingleRow* - se returnează o singură linie. De exemplu, dacă în codul anterior înainte de *while* obţinerea obiectului *reader* s–ar face cu:

  ```
  SqlDataReader reader = cmd.ExecuteReader(
     CommandBehavior.SingleRow);
  ```

  atunci s–ar returna doar prima înregistrare din setul de date.

6. *ExecuteScalar()* - execută comanda conţinută în proprietatea *CommandText*; se returnează valoarea primei coloane de pe primul rând a setului de date rezultat; folosit pentru obţinerea unor rezultate de tip agregat ("SELECT COUNT(*) FROM CUSTOMERS", de exemplu).

7. *ExecuteXmlReader()* - returnează un obiect de tipul *XmlReader* obţinut din rezultatul interogării pe sursa de date.

   Exemplu:

```
SqlCommand custCMD=new SqlCommand("SELECT * FROM Customers
   FOR XML AUTO, ELEMENTS", con);
System.Xml.XmlReader myXR = custCMD.ExecuteXmlReader();
```

8.5.3 Utilizarea unei comenzi cu o procedură stocată

Pentru a se executa pe server o procedură stocată definită în baza respectivă, este necesar ca obiectul comandă să aibe proprietatea *CommandType* la valoarea *CommandType.StoredProcedure* iar proprietatea *CommandText* să conţină numele procedurii stocate:

```
SqlConnection con = new SqlConnection(
  ConfigurationManager.ConnectionStrings["constring"]
    .ConnectionString);
SqlCommand cmd = new SqlCommand("Ten Most Expensive Products", con);
cmd.CommandType = CommandType.StoredProcedure;
con.Open();
SqlDataReader reader = cmd.ExecuteReader();
while(reader.Read())
{
 Console.WriteLine("{0} - {1}",
 reader.GetString(0), reader.GetDecimal(1));
}
reader.Close();
con.Close();
```

Observaţie: în toate exemplele de mai sus faptul fiecare conexiune se închide manual, printr-un apel de tipul `con.Close()`. Daca conexiunea a fost folosită pentru un obiect de tip *DataRead* atunci acesta din urmă trebuie să fie şi el închis, înaintea închiderii conexiunii. Dacă nu se face acest apel atunci conexiunea va fi ţinută ocupată şi nu va putea fi reutilizată.

8.5.4 Folosirea comenzilor parametrizate

Există posibilitatea de a rula cod SQL (interogări sau proceduri stocate) pe server care să fie parametrizate. Orice furnizor de date .NET permite crearea obiectelor parametru care pot fi adăugate la o colecţie de parametri

ai comenzii. Valoarea acestor parametri se specifică fie prin numele lor (cazul *SqlParameter*), fie prin poziţia lor (cazul *OleDbParameter*).

Exemplu: vom aduce din tabela *Customers* toate înregistrările care au în câmpul *Country* valoarea "USA".

```
SqlConnection con = new SqlConnection(
ConfigurationManager.ConnectionStrings["constring"]
  .ConnectionString);
SqlCommand cmd = new
  SqlCommand("SELECT * FROM Customers WHERE Country=@country",con);
SqlParameter param = new SqlParameter("@country", SqlDbType.VarChar);
param.Value = "USA";
cmd.Parameters.Add( param );
con.Open();
SqlDataReader reader = cmd.ExecuteReader();
while(reader.Read())
{
  Console.WriteLine("{0} - {1}",
   reader.GetString(0), reader.GetString(1));
}
reader.Close();
con.Close();
```

Pentru parametrul creat s–a setat tipul lui (ca fiind tip şir de caractere SQL) şi valoarea. De reţinut faptul că numele parametrului se prefixează cu caracterul "@" în cazul lucrului cu SQL Server.

În cazul în care un parametru este de ieşire, acest lucru trebuie spus explicit folosind proprietatea *Direction* a parametrului respectiv:

```
SqlCommand cmd = new SqlCommand(
 "SELECT * FROM Customers WHERE Country = @country; " +
 "SELECT @count = COUNT(*) FROM Customers WHERE Country = @country",
 con);
SqlParameter param = new SqlParameter("@country", SqlDbType.VarChar);
param.Value = "USA";
cmd.Parameters.Add( param );
cmd.Parameters.Add(new SqlParameter("@count", SqlDbType.Int));
cmd.Parameters["@count"].Direction = ParameterDirection.Output;
con.Open();
SqlDataReader reader = cmd.ExecuteReader();
while(reader.Read())
{
```

```
  Console.WriteLine("{0} - {1}",
  reader.GetString(0),
  reader.GetString(1));
}
reader.Close();
Console.WriteLine("{0} - {1}", "Count",
  cmd.Parameters["@count"].Value.ToString());
con.Close();
```

Remarcăm următoarele:

- este posibil ca într–o comandă să se execute mai multe interogări
- pentru parametrul de ieşire numit "@count" trebuie făcută declarare de direcţie; implicit un parametru este de intrare
- parametrii de ieşire sunt accesibili doar după închiderea obiectului de tip *DataReader*

8.6 Obiecte *DataReader*

Un obiect de tip *DataReader* este folosit pentru a citi date dintr-o sursă de date. Caracteristicile unei asemenea clase sunt:

1. implementează întotdeauna interfaţa *IDataReader*
2. se lucrează conectat la sursa de date - pe toată perioada cât este accesat un *DataReader* necesită conexiune activă
3. este de tip read-only; dacă se doreşte modificarea datelor se poate folosi un *DataSet* + *DataAdapter* sau comenzi INSERT, DELETE sau UPDATE trimise prin obiect de tip *Command*;
4. este de tip forward-only - metoda de modificare a poziţiei curente este doar în direcţia înainte; orice reîntoarcere presupune reluarea înregistrărilor (dacă programatorul nu implementează el singur un mecanism de coadă)

Avantajele utilizării acestui tip de obiecte sunt: accesul conectat, performanţele bune, consumul mic de resurse şi tipizarea puternică.

8.6.1 Proprietăţi

1. *IsClosed* - proprietate read-only, returnează *true* daca obiectul este deschis, *false* altfel

2. *HasRows* - proprietate booleană read-only care spune dacă readerul conţine cel puţin o înregistrare

3. *Item* - indexator care dă acces la câmpurile unei înregistrări

4. *FieldCount* - dă numărul de câmpuri din înregistrarea curentă

8.6.2 Metode

1. *Close()* - închide obiectul de citire şi eliberează resursele client. Este obligatoriu apelul acestei metode înaintea închiderii conexiunii.

2. *GetBoolean(), GetByte(), GetChar(), GetDateTime(), GetDecimal(), GetDouble(), GetFloat(), GetInt16(), GetInt32(), GetInt64(), GetValue(), GetString()* returnează valorile câmpurilor din înergistrarea curentă. Preiau ca parametru indicele coloanei a cărei valoare se cere. *GetValue()* returnează un obiect de tip *Object*, pentru celelalte tipul returnat este descris de numele metodelor.

3. *GetBytes(), GetChars()* - returnează numărul de octeţi / caractere citiţi dintr–un câmp ce stochează o structură de dimensiuni mari; primeşte ca parametri indicele de coloană (int), poziţia din acea coloană de unde se va începe citirea, vectorul în care se face citirea, poziţia în buffer de la care se depun datele citite, numărul de octeţi/caractere ce urmează a fi citiţi.

4. *GetDataTypeName()* - returnează tipul coloanei specificat prin indice

5. *GetName()* - returnează numele coloanei

6. *IsDBNull()* - returnează *true* dacă în câmpul specificat prin index este o valoare de *NULL* (din baza de date)

7. *NextResult()* - determină trecerea la următorul rezultat, dacă aceasta există; în acest caz returnează *true*, altfel *false* (este posibil ca într–un *DataReader* să vină mai multe rezultate, provenind din interogări diferite)

8. *Read()* - determină trecerea la următoarea înregistrare, dacă aceasta există; în acest caz ea returnează *true*. Metoda trebuie chemată cel puţin o dată, deoarece iniţial poziţia curentă este înaintea primei înregistrări.

8.6.3 Crearea şi utilizarea unui *DataReader*

Nu se crează un obiect de tip *DataReader* prin apel de constructor, ci prin intermediul unui obiect de tip *Command*, folosind apelul *ExecuteReader()* (a se vedea secţiunea 8.3.2). Pentru comanda respectivă se specifică instrucţiunea care determină returnarea setului de date precum şi obiectul de conexiune. Această conexiune trebuie să fie deschisă înaintea apelului *ExecuteReader()*. Trecerea la următoarea înregistrare se face folosind metoda *Read()*. Când se doreşte încetarea lucrului se închide reader–ul şi conexiunea. Omiterea închiderii obiectului de tip reader va duce la imposibilitatea reutilizării conexiunii iniţiale. După ce se închide acest *DataReader* este necesară şi în chiderea explicită a conexiunii (acest lucru nu mai e mandatoriu doar dacă la apelul metodei *ExecuteReader* s–a specificat *CommandBehavior.CloseConnection*). Dacă se încearcă refolosirea conexiunii fără ca readerul să fi fost închis se va arunca o excepţie *InvalidOperationException*.

Exemplu:

```
SqlConnection conn = new SqlConnection (
  ConfigurationManager.ConnectionStrings["constring"]
    .ConnectionString);
SqlCommand selectCommand = new SqlCommand("select * from ORDERS", conn);
conn.Open ();
OleDbDataReader reader = selectCommand.ExecuteReader ( );
while ( reader.Read () )
{
  object id = reader["OrderID"];
  object date = reader["OrderDate"];
  object freight = reader["Freight"];
  Console.WriteLine ( "{0}\t{1}\t\t\t{2}", id, date, freight );
}
reader.Close ();
conn.Close ();
```

Este perfect posibil ca un obiect de tip *DataReader* să aducă datele prin apelul unei proceduri stocate (de fapt invocarea acestei proceduri este făcută de către obiectul de tip *Command*).

Următoarele observaţii trebuie luate în considerare atunci când se lucrează cu un obiect *DataReader*:

- Metoda *Read()* trebuie să fie întotdeauna apelată înaintea oricărui acces la date; poziţia curentă la deschidere este înaintea primei înregistrări.
- Întotdeauna apelaţi metoda *Close()* pe un *DataReader* şi pe conexiunea asociată cât mai repede posibil; în caz contrar conexiunea nu poate fi reutilizată
- Procesarea datelor citite trebuie să se facă după închiderea conexiunii; în felul acesta conexiunea se lasă liberă pentru a putea fi reutilizată.

8.6.4 Utilizarea de seturi de date multiple

Este posibil ca într–un *DataReader* să se aducă mai multe seturi de date. Acest lucru ar micşora numărul de apeluri pentru deschiderea de conexiuni la stratul de date. Obiectul care permite acest lucru este chiar cel de tip *Command*:

```
string select = "select * from Categories; select * from customers";
SqlCommand command = new SqlCommand ( select, conn );
conn.Open ();
SqlDataReader reader = command.ExecuteReader ();
```

Trecerea de la un set de date la altul se face cu metoda *NextResult()* a obiectului de tip *Reader*:

```
do
{
  while ( reader.Read () )
  {
    Console.WriteLine ( "{0}\t\t{1}", reader[0], reader[1] );
  }
}while ( reader.NextResult () );
```

8.6.5 Accesarea datelor într–o manieră sigură din punct de vedere a tipului

Să considerăm următoarea secvenţă de cod:

```
while ( reader.Read () )
{
```

```
  object id = reader["OrderID"];
  object date = reader["OrderDate"];
  object freight = reader["Freight"];
  Console.WriteLine ( "{0}\t{1}\t\t{2}", id, date, freight );
}
```

După cum se observă, este posibil ca valorile câmpurilor dintr–o înregistrare să fie accesate prin intermediul numelui coloanei (sau a indicelui ei, pornind de la 0). Dezavantajul acestei metode este că tipul datelor returnate este pierdut (fiind returnate obiecte de tip *Object*), trebuind făcută un downcasting pentru a utiliza din plin facilităţile tipului respectiv. Pentru ca acest lucru să nu se întâmple se pot folosi metodele *GetXY* care returnează un tip specific de date:

```
while ( reader.Read () )
{
  int id = reader.GetInt32 ( 0 );
  DateTime date = reader.GetDateTime ( 3 );
  decimal freight = reader.GetDecimal ( 7 );
  Console.WriteLine ( "{0}\t{1}\t\t{2}", id, date, freight );
}
```

Avantajul secvenţei anterioare este că dacă se încearcă aducerea valorii unui câmp pentru care tipul nu este dat corect se aruncă o excepţie *InvalidCastException*; altfel spus, accesul la date se face sigur din punct de verere al tipului datelor.

Pentru a evita folosirea unor "constante magice" ca indici de coloană (precum mai sus: 0, 3, 7), se poate folosi următoarea strategie: indicii se obţin folosind apel de metodă *GetOrdinal* la care se specifică numele coloanei dorite:

```
private int OrderID;
private int OrderDate;
private int Freight;
...
OrderID = reader.GetOrdinal("OrderID");
OrderDate = reader.GetOrdinal("OrderDate");
Freight = reader.GetOrdinal("Freight");
...
reader.GetDecimal ( Freight );
...
```

Curs 9

ADO.NET (2)

9.1 Obiecte *DataAdapter*

La fel ca şi *Connection*, *Command*, *DataReader*, obiectele de tip *DataAdapter* fac parte din furnizorul de date .NET specific fiecărui tip de sursă de date. Scopul clasei este să permită umplerea unui obiect *DataSet* cu date şi reflectarea schimbărilor efectuate asupra acestuia înapoi în baza de date (*DataSet* permite lucrul deconectat de la baza de date).

Orice clasă de tipul *DataAdapter* (de ex SqlDataAdapter şi OleDbDataAdapter) este derivată din clasa *DbDataAdapter* (clasă abstractă). Pentru orice obiect de acest tip trebuie specificată minim comanda de tip *SELECT* care să populeze un obiect de tip *DataSet*; acest lucru este stabilit prin intermediul proprietăţii *SelectCommand* de tip *Command* (*SqlCommand*, *OleDbCommand*, ...). În cazul în care se doreşte şi modificarea informaţiilor din sursa de date (inserare, modificare, ştergere) trebuie specificate obiecte de tip comandă via proprietăţile: *InsertCommand*, *UpdateCommand*, *DeleteCommand*.

Exemplu: mai jos se preiau înregistrările din 2 tabele: *Authors* şi *TitleAuthor* şi se trec într–un obiect de tip *DataSet* pentru a fi procesate ulterior.

```
using System;
using System.Data;
using System.Data.SqlClient;

class DemoDataSource
{
  static void Main()
  {
    SqlConnection conn = new SqlConnection(
      ConfigurationManager.ConnectionStrings["constring"]
```

```
        .ConnectionString);
    DataSet ds = new DataSet();
    SqlDataAdapter daAuthors = new SqlDataAdapter("SELECT au_id,
      au_fname, au_lname FROM authors",conn);
    daAuthors.Fill(ds,"Author");

    SqlDataAdapter daTitleAuthor = new SqlDataAdapter("SELECT
      au_id, title_id FROM titleauthor", conn);
    daTitleAuthor.Fill(ds,"TitleAuthor");
  }
}
```

Prezentăm mai jos cele mai importante componente ale unei clase de tip *DataAdapter*.

9.1.1 Metode

1. *Constructori* de la cei impliciţi până la cei în care se specifică o comandă de tip *SELECT* şi conexiunea la sursa de date. Pentru un obiect de tip *SqlDataAdapter* se poate crea o instanţă în următoarele moduri:

   ```
   SqlDataAdapter da = new SqlDataAdapter();
   ```

   sau:

   ```
   SqlCommand cmd = new SqlCommand("SELECT * FROM Employees");
   SqlDataAdapter da = new SqlDataAdapter(cmd);
   ```

   sau:

   ```
   String strCmd = "SELECT * FROM Employees";
   String strConn = "...";
   SqlDataAdapter da = new SqlDataAdapter(strCmd, strConn);
   ```

2. *Fill()* – metodă polimorfică, permiţând umplerea unei tabele dintr–un obiect de tip *DataSet* cu date. Permite specificarea obiectului *DataSet* în care se depun datele, eventual a numelui tablei din acest *DataSet*, numărul de înregistrare cu care să se înceapă popularea (prima având indicele 0) şi numărul de înregistrări care urmează a fi aduse. Returnează de fiecare dată numărul de înregistrări care au fost aduse din bază. În clipa în care se apelează *Fill()* se procedează astfel:

(a) Se deschide conexiunea (dacă ea nu a fost explicit deschisă)

(b) Se aduc datele şi se populează un obiect de tip *DataTable* din *DataSet*

(c) Se închide conexiunea (dacă ea nu a fost explicit deschisă!)

De remarcat că un *DataAdapter* îşi poate deschide şi închide singur conexiunea, dar dacă aceasta a fost deschisă înaintea metodei *Fill()* atunci tot programatorul trebuie să o închidă.

3. *Update()* – metodă polimorfică, permiţând reflectarea modificărilor efectuate între–un *DataSet*. Pentru a funcţiona are nevoie de obiecte de tip comandă adecvate: proprietăţile *InsertCommand*, *DeleteCommand* şi *UpdateCommand* trebuie să indice către comenzi valide. Returnează de fiecare dată numărul de înregistrări afectate.

9.1.2 Proprietăţi

1. *DeleteCommand*, *InsertCommand*, *SelectCommand*, *UpdateCommand* – de tip *Command*, conţin comenzile ce se execută pentru selectarea sau modificarea datelor în sursa de date. Măcar proprietatea *SelectCommand* trebuie să indice către un obiect valid, pentru a se putea face popularea setului de date.

2. *MissingSchemaAction* – de tip enumerare *MissingSchemaAction*, determină ce se face atunci când datele care sunt aduse nu se potrivesc peste schema tablei în care sunt depuse. Poate avea următoarele valori:

   - *Add* - implicit, *DataAdapter* adaugă coloana la schema tablei
   - *AddWithKey* - ca mai sus, dar adaugă şi informaţii relativ la cine este cheia primară
   - *Ignore* - se ignoră lipsa coloanei respective, ceea ce duce la pierdere de date pe *DataSet*
   - *Error* - se generează o excepţie de tipul *InvalidOperationException*.

9.2 Clasa *DataSet*

Clasa *DataSet* nu mai face parte din biblioteca unui furnizor de date ADO.NET, fiind parte din .NET Framework. Ea poate să conţină reprezentări tabelare ale datelor din bază precum şi diferite restricţii şi relaţii existente. Marele ei avantaj este faptul că permite lucrul deconectat de la sursa de date,

eliminând necesitatea unei conexiuni permanente precum la *DataReader*. În felul acesta, un server de aplicaţii sau un client oarecare poate apela la serverul de baze de date doar când preia datele sau când se doreşte salvarea lor. Funcţionează în strânsă legătură cu clasa *DataAdapter* care acţionează ca o punte între un *DataSet* şi sursa de date. Remarcabil este faptul că un *DataSet* poate face abstracţie de sursa de date, procesarea datelor desfăşurându–se independent de ea.

Figura 9.1 conţine o vedere parţială asupra clasei *DataSet*.

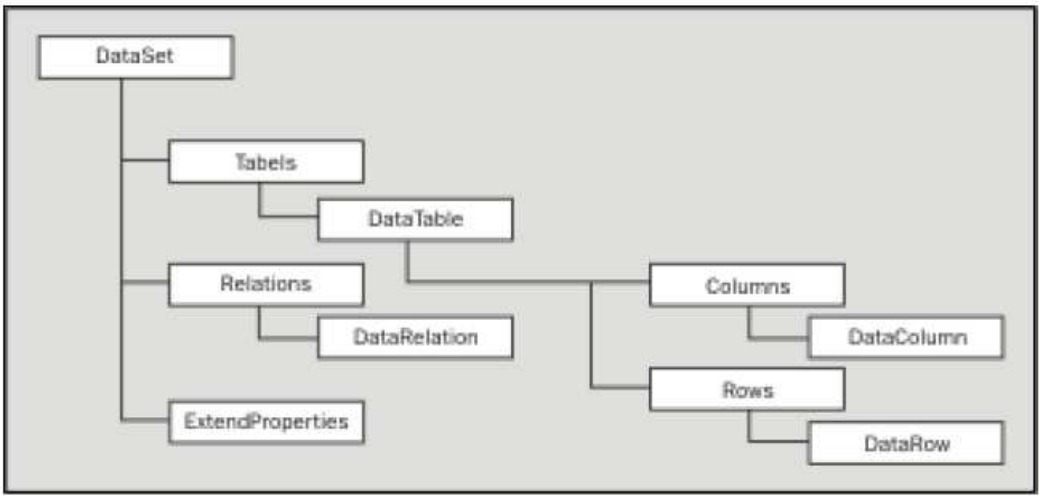

Figura 9.1: Structura unui *DataSet*

9.2.1 Conţinut

Prezentăm succint conţinutul unui *DataSet*:

1. Colecţia *Tables* conţine 0 sau mai multe obiecte *DataTable*. Fiecare *DataTable* este compusă dintr-o colecţie de linii şi coloane.

2. Colecţia *Relations* conţine 0 sau mai multe obiecte de tip *DataRelation*, folosite pentru marcarea legăturilor părinte–copil.

3. Colecţia *ExtendedProperties* conţine proprietăţi definite de utilizator.

9.2.2 Clasa *DataTable*

Datele sunt conţinute într-un *DataSet* sub forma unor tabele de tip *DataTable*. Aceste obiecte pot fi folosite atât independent, cât şi în interiorul unui

DataSet ca elemente ale colecţiei *Tables*. Un *DataTable* conţine o colecţie *Columns* de coloane, *Rows* de linii şi *Constraints* de constrângeri.

DataColumn

Un obiect *DataColumn* defineşte numele şi tipul unei coloane care face parte sau se adaugă unui obiect *DataTable*. Un obiect de acest tip se obţine prin apel de constructor sau pe baza metodei *DataTable.Columns.Add*.

Exemplu:

```
DataColumn myColumn = new DataColumn("title",
  Type.GetType("System.String"));
```

Definirea unei coloane ca fiind de tip autonumber (în vederea stabilirii ei ca şi cheie pe o tabelă) se face astfel:

```
DataColumn idColumn = new DataColumn("ID",
  Type.GetType("System.Int32"));
idColumn.AutoIncrement = true;
idColumn.AutoIncrementSeed = 1;
idColumn.AutoIncrementStep = 1;
idColumn.ReadOnly = true;
```

DataRow

Un obiect de tip *DataRow* reprezintă o linie dintr-un obiect *DataTable*. Orice obiect *DataTable* conţine o proprietate *Rows* ce da acces la colecţia de obiecte *DataRow* conţinută. Pentru crearea unei linii se poate apela metoda *NewRow* pentru o tabelă a cărei schemă se cunoaşte. Mai jos este dată secvenţa de cod care creează o linie nouă pentru o tabelă şi o adaugă acestesia:

```
DataRow tempRow;
tempRow = myTable.NewRow();
tempRow["Name"] = "Book";
tempRow["Category"] = 1;
myTable.Rows.Add(tempRow);
```

Constrângeri

Constrângerile sunt folosite pentru a descrie anumite restricţii aplicate asupra valorilor din coloane. În ADO.NET există două tipuri de constrângeri: de unicitate şi de cheie străină. Toate obiectele de constrângere se află în colecţia *Constraints* a unei tabele.

- *UniqueConstraint* - precizează că într-o anumită coloană valorile sunt unice. Încercarea de a seta valori duplicate pe o coloana pentru care s-a precizat restricţia duce la aruncarea unei excepţii. Este necesară o asemenea coloană în clipa în care se foloseşte metoda *Find* pentru proprietatea *Rows*: în acest caz trebuie să se specifice o coloană pe care avem unicitate.

- *ForeignKeyConstraint* - specifică acţiunea care se va efectua atunci când se şterge sau modifică valoarea dintr–o anumită coloană. De exemplu se poate decide că dacă se sterge o înregistrare dintr-o tabelă atunci să se şteargă şi înregistrările copil. Valorile care se pot seta pentru o asemenea constrângere se specifică în proprietăţile *ForeignKeyConstraint.Delet* şi *ForeignKeyConstraint.UpdateRule*:

  – *Rule.Cascade* - acţiunea implicită, şterge sau modifică înregistrările afectate
  – *Rule.SetNull* - se setează valoare de **null** pentru înregistrările afectate
  – *Rule.SetDefault* - se setează valoarea implicită definită în bază pentru câmpul respectiv
  – *Rule.None* - nu se execută nimic

  Exemplu:

```
ForeignKeyConstraint custOrderFK=new ForeignKeyConstraint
 ("CustOrderFK",custDS.Tables["CustTable"].Columns["CustomerID"],
  custDS.Tables["OrdersTable"].Columns["CustomerID"]);
custOrderFK.DeleteRule = Rule.None;
//Nu se poate sterge un client care are comenzi facute
custDS.Tables["OrdersTable"].Constraints.Add(custOrderFK);
```

  Mai sus s-a declarat o relaţie de tip cheie străină între două tabele (“CustTable” şi “OrdersTable”, care fac parte dintr-un DataSet). Restricţia se adaugă la tabla copil.

Stabilirea cheii primare

O cheie primară se defineşte ca un vector de coloane care se atribuie proprietăţii *PrimaryKey* a unei tabele (obiect *DataTable*).

```
DataColumn[] pk = new DataColumn[1];
pk[0] = myTable.Columns["ID"];
myTable.PrimaryKey = pk;
```

Proprietatea *Rows* a clasei *DataTable* permite căutarea unei anumite linii din colecţia conţinută dacă se specifică un obiect sau un array de obiecte folosite pe post de cheie:

```
object key = 17;//cheia dupa care se face cautarea
DataRow line = myTable.Rows.Find(key);
if ( line != null )
  //proceseaza linia
```

9.2.3 Relaţii între tabele

Proprietatea *Relations* a unui obiect de tip *DataSet* conţine o colecţie de obiecte de tip *DataRelation* folosite pentru a figura relaţiile de tip părinte–copil între două tabele. Aceste relaţii se precizează în esenţă ca niste perechi de array-uri de coloane sau chiar coloane simple din cele două tabele care se relaţionează, de exemplu sub forma:

```
myDataSet.Relations.Add(DataColumn, DataColumn);
//sau
myDataSet.Relations.Add(DataColumn[], DataColumn[]);
```

concret:

```
myDataSet.Relations.Add(
  myDataSet.Tables["Customers"].Columns["CustomerID"],
  myDataSet.Tables["Orders"].Columns["CustomerID"]);
```

9.2.4 Popularea unui *DataSet*

Deşi un obiect *DataSet* se poate popula prin crearea dinamică a obiectelor *DataTable*, cazul cel mai des întâlnit este acela în care se populează prin intermediul unui obiect *DataAdapter*. O dată obţinut un asemenea obiect (care conţine cel puţin o comandă de tiop *SELECT*) se poate apela metoda *Fill()* care primeşte ca parametru *DataSet*-ul care se umple şi opţional numele tabelei care va conţine datele:

```
//defineste comanda de selectare din baza de date
String mySqlStmt ="SELECT * FROM Customers";
String myConString = ConfigurationManager.ConnectionStrings["constring"]
  .ConnectionString;
//Construieste obiectele de conexiune + comanda SELECT
SqlConnection myConnection = new SqlConnection(myConString);
SqlCommand myCommand = new SqlCommand(mySqlStmt, myConnection);
```

```
//Construieste obiectul DataAdapter
SqlDataAdapter myDataAdapter = new SqlDataAdapter();
//seteaza proprietatea SelectCommand pentru DataAdapter
myDataAdapter.SelectCommand = myCommand;
//construieste obiectul DataSet si il umple cu date
DataSet myDataSet = new DataSet();
myDataAdapter.Fill(myDataSet, "Customers");
```

Datele aduse mai sus sunt depuse într-un obiect de tip *DataTable* din interiorul lui *DataSet*, numit "Customers". Accesul la acest tabel se face prin construcţia

```
myDataSet.Tables["Customers"]
```

sau folosind indici întregi (prima tabelă are indicele 0). Acelaşi *DataSet* se poate popula în continuare cu alte tabele pe baza aceluiaşi sau a altor obiecte *DataAdapter*.

9.2.5 Clasa *DataTableReader*

Începând cu versiunea 2.0 a lui ADO.NET s-a introdus clasa *DataTableReader* care permite manipularea unui obiect de tip *DataTable* ca şi cum ar fi un *DataReader*: într-o manieră *forward-only* şi *read-only*. Crearea unui obiect de tip *DataTableReader* se face prin:

```
DataTableReader dtReader = dt.CreateDataReader();
```

iar folosirea lui:

```
while (dtReader.Read())
{
  for (int i = 0; i < dtReader.FieldCount; i++)
  {
    Console.Write("{0} = {1} ",
    dtReader.GetName(i),
    dtReader.GetValue(i).ToString().Trim());
  }
  Console.WriteLine();
}
dtReader.Close();
```

9.2.6 Propagarea modificărilor către baza de date

Pentru a propaga modificările efectuate asupra conţinutului tabelelor dintr-un *DataSet* către baza de date este nevoie să se definească adecvat obiecte comandă de tip *INSERT*, *UPDATE*, *DELETE*. Pentru cazuri simple se poate folosi clasa *SqlCommandBuilder* care va construi singură aceste comenzi.

Clasa *CommandBuilder*

Un obiect de tip *CommandBuilder* (ce provine din furnizorul de date) va analiza comanda *SELECT* care a adus datele în *DataSet* şi va construi cele 3 comenzi de update în funcţie de aceasta. E nevoie să se satisfacă 2 condiţii atunci când se uzează de un astfel de obiect:

1. Trebuie specificată o comandă de tip *SELECT* care să aducă datele dintr-o singură tabelă

2. Trebuie specificată cel puţin cheia primară sau o coloană cu constrângere de unicitate în comanda *SELECT*.

Pentru cea de a doua condiţie se poate proceda în felul următor: în comanda *SELECT* se specifică şi aducerea cheii, iar pentru obiectul *DataAdapter* care face aducerea din bază se setează proprietatea *MissingSchemaAction* pe valoarea *MissingSchemaAction.AddWithKey* (implicit este doar *Add*).

Fiecare linie modificată din colecţia *Rows* a unei tabele va avea modificată valoarea proprietăţii *RowState* astfel: *DataRowState.Added* pentru o linie nouă adăugată, *DataRowState.Deleted* dacă e ştearsă şi *DataRowState.Modified* dacă a fost modificată. Apelul de update pe un dataReader va apela comanda necesară pentru fiecare linie care a fost modificată, în funcţie de starea ei.

Arătăm mai jos modul de utilizare a clasei *SqlCommandBuilder* pentru adăugarea, modificarea, ştergerea de înregistrări din baza de date.

```
SqlConnection conn = new SqlConnection(
ConfigurationManager.ConnectionStrings["constring"]
  .ConnectionString);
da = new SqlDataAdapter("SELECT id, name, address FROM
  customers",conn);
da.MissingSchemaAction = MissingSchemaAction.AddWithKey;
da.Fill(ds);

SqlCommandBuilder cb = new SqlCommandBuilder(da);
//determina liniile care au fost schimbate
```

```
DataSet dsChanges = ds.GetChanges();
if (dsChanges != null)
{
  // modifica baza de date
  da.Update(dsChanges);
  //accepta schimbarile din dataset
  ds.AcceptChanges();
}
```

În clipa în care se creează obiectul *SqlCommandBuilder* automat se vor completa proprietăţile *InsertCommand*, *DeleteCommand*, *UpdateCommand* ale dataAdapter-ului. Se determină apoi liniile care au fost modificate (prin interogarea stării lor) şi se obţine un nou *DataSet* care le va conţine doar pe acestea. Comanda de *Update* se dă doar pentru acest set de modificări, reducând astfel traficul spre serverul de baze de date.

Update folosind comenzi SQL

Atunci când interogările de aducere a datelor sunt mai complexe (de exemplu datele sunt aduse din mai multe table, printr-un join) se pot specifica propriile comenzi SQL prin intermediul proprietăţilor *InsertCommand*, *DeleteCommand* *UpdateCommand* ale obiectului *DataAdapter*. Pentru fiecare linie dintr-o tabelă care este modificată/adăugată/ştearsă se va apela comanda SQL corespunzătoare.Aceste comenzi pot fi fraze SQL parametrizate sau pot denumi proceduri stocate aflate în bază.

Să presupunem că s-a definit un *DataAdapter* legat la o bază de date. Instrucţiunea de selecţie este

```
SELECT CompanyName, Address, Country, CustomerID FROM Customers
```

unde *CustomerID* este cheia. Pentru inserarea unei noi înregistrări s–ar putea scrie codul de mai jos:

```
//da=obiect DataAdapter
da.InsertCommand.CommandText = @"INSERT INTO Customers (
  CompanyName, Address, Country) VALUES
  (@CompanyName, @Address, @Country);
  SELECT CompanyName, Address, Country, CustomerID FROM
  Customers WHERE (CustomerID = scope_indentity())";
```

Update-ul efectiv se face prin prima instrucţiune de tip *Update*. Valorile pentru aceşti parametri se vor da la runtime, de exemplu prin alegerea lor dintr-un tabel. Valoarea pentru cheia CustomerID nu s-a specificat,

deoarece (în acest caz) ea este de tip AutoNumber (SGBD-ul este cel care face managementul valorilor acestor câmpuri, nu programatorul). *scope_ indentity()* este o funcţie predefinită ce returnează id-ul noii înregistrări adăugate în tabelă. Ultima instrucţiune va duce la reactualizarea obiectului *DataSet*, pentru ca acesta să conţină modificarea efectuată (de exemplu ar putea aduce valorile implicite puse pe anumite coloane).

Pentru modificarea conţinutului unei linii se poate declara instrucţiunea de *UPDATE* astfel:

```
da.UpdateCommand.CommandText = @"UPDATE Customers SET CompanyName
 = @CompanyName, Address = @Address, Country = @Country
 WHERE (CustomerID = @ID)";
```

9.3 Tranzacţii în ADO.NET

O tranzacţie este un set de operaţii care se efectuează fie în întregime, fie deloc. Să presupunem că se doreşte trecerea unei anumite sume de bani dintr-un cont în altul. Operaţia presupune 2 paşi:

1. scade suma din primul cont
2. adaugă suma la al doilea cont

Este inadmisibil ca primul pas să reuşească iar al doilea să eşueze. Tranzacţiile satisfac nişte proprietăţi strânse sub numele ACID:

- **atomicitate** - toate operaţiile din tranzacţie ar trebui să aibă succes sau să eşueze împreună
- **consistenţă** - tranzacţia duce baza de date dintr-o stare stabilă în alta
- **izolare** - nici o tranzacţie nu ar trebui să afecteze o alta care rulează în acelaşi timp
- **durabilitate** - schimbările care apar în tipul tranzacţiei sunt permanent stocate pe un mediu.

Sunt trei comenzi care se folosesc în context de tranzacţii:

- *BEGIN* - înainte de executarea unei comenzi SQL sub o tranzacţie, aceasta trebuie să fie iniţializată
- *COMMIT* - se spune că o tranzacţie este terminată când toate schimbările cerute sunt trecute în baza de date

- *ROLLBACK* - dacă o parte a tranzacţiei eşuează, atunci toate operaţiile efectuate de la începutul tranzacţiei vor fi neglijate

Schema de lucru cu tranzacţiile sub ADO.NET este:

1. deschide conexiunea la baza de date
2. începe tranzacţia
3. execută comenzi pentru tranzacţie
4. dacă tranzacţia se poate efectua (nu sunt excepţii sau anumite condiţii sunt îndeplinite), efectuează *COMMIT*, altfel efectuează *ROLLBACK*
5. închide conexiunea la baza de date

Sub ADO.NET acest lucru s-ar face astfel:

```
SqlConnection myConnection = new SqlConnection(myConnString);
myConnection.Open();

SqlCommand myCommand1 = myConnection.CreateCommand();
SqlCommand myCommand2 = myConnection.CreateCommand();
SqlTransaction myTrans;

myTrans = myConnection.BeginTransaction();
//Trebuie asignate ambele obiecte: conexiune si tranzactie
//unui obiect de tip comanda care va participa la tranzactie
myCommand1.Transaction = myTrans;
myCommand2.Transaction = myTrans;

try
{
  myCommand1.CommandText = "Insert into Region (RegionID,
    RegionDescription) VALUES (100, 'Description')";
  myCommand1.ExecuteNonQuery();
  myCommand2.CommandText = "Insert into Region (RegionID,
    RegionDescription) VALUES (101, 'Description')";
  myCommand2.ExecuteNonQuery();
  myTrans.Commit();
  Console.WriteLine("Ambele inregistrari au fost scrise.");
}
catch(Exception e)
```

```
{
  myTrans.Rollback();
}
finally
{
  myConnection.Close();
}
```

Comanda de *ROLLBACK* se poate executa şi în alte situaţii, de exemplul comanda efectuată depăşteşte stocul disponibil.

9.4 Lucrul generic cu furnizori de date

În cele expuse până acum, s-a lucrat cu un furnizor de date specific pentru SQL Server 2005. În general e de dorit să se scrie cod care să funcţioneze fără modificări majore pentru orice furnizor de date; mai exact, am prefera să nu fie nevoie de rescrierea sau recompilarea codului. Începând cu versiunea 2.0 a lui ADO.NET se poate face acest lucru uşor, prin intermediul unei clase *DbProviderFactory* (un *Abstract factory*).

Mecanismul se bazează pe faptul că avem următoarele clase de bază pentru tipurile folosite într-un furnizor de date:

- *DbCommand*: clasă de bază abstractă pentru obiectele de tip *Command*
- *DbConnection*: clasă de bază abstractă pentru obiectele de tip *Connection*
- *DbDataAdapter*: clasă de bază abstractă pentru obiectele de tip *DataAdapter*
- *DbDataReader*: clasă de bază abstractă pentru obiectele de tip *DataReader*
- *DbParameter*: clasă de bază abstractă pentru obiectele de tip parametru
- *DbTransaction*: clasă de bază abstractă pentru obiectele de tip tranzacţie

Crearea de obiecte specifice (de exemplu obiect *SqlCommand*) se face folosind clase derivate din *DbProviderFactory*; o schiţă a acestei clase este:

```
public abstract class DbProviderFactory
{
...
  public virtual DbCommand CreateCommand();
  public virtual DbCommandBuilder CreateCommandBuilder();
  public virtual DbConnection CreateConnection();
```

```
  public virtual DbConnectionStringBuilder CreateConnectionStringBuilder();
  public virtual DbDataAdapter CreateDataAdapter();
  public virtual DbDataSourceEnumerator CreateDataSourceEnumerator();
  public virtual DbParameter CreateParameter();
}
```

Tot ceea ce trebuie făcut este să se obţină o clasă concretă derivată din *DbProviderFactory* şi care la apeluri de tip *Create...* să returneze obiecte concrete, adecvate pentru lucrul cu sursa de date. Concret:

```
static void Main(string[] args)
{
  // Obtine un producator pentru SqlServer
  DbProviderFactory sqlFactory =
    DbProviderFactories.GetFactory("System.Data.SqlClient");
  ...
  // Obtine un producator pentru Oracle
  DbProviderFactory oracleFactory =
    DbProviderFactories.GetFactory("System.Data.OracleClient");
  ...
}
```

Se va evita, fireste, codificarea numelui furnizorului de date în cod (precum mai sus) si se vor integra în fişiere de configurare. Aceste şiruri de caractere exemplificate mai sus sunt definite în fişierul machine.config din directorul unde s-a făcut instalarea de .NET (%windir%\Microsoft.Net\Framework\v2.0.50727\config).

Exemplu: fişierul de configurare este:

```
<configuration>
<appSettings>
<!-- Provider -->
<add key="provider" value="System.Data.SqlClient" />
<!-- String de conexiune -->
<add key="cnStr" value=
"Data Source=localhost;uid=sa;pwd=;Initial Catalog=Pubs"/>
</appSettings>
</configuration>
```

Codul C#:

```
static void Main(string[] args)
{
```

```
string dp = ConfigurationManager.AppSettings["provider"];
string cnStr = ConfigurationManager.ConnectionStrings["constring"]
  .ConnectionString;

DbProviderFactory df = DbProviderFactories.GetFactory(dp);
DbConnection cn = df.CreateConnection();
cn.ConnectionString = cnStr;
cn.Open();

DbCommand cmd = df.CreateCommand();
cmd.Connection = cn;
cmd.CommandText = "Select * From Authors";

DbDataReader dr = cmd.ExecuteReader(CommandBehavior.CloseConnection);

while (dr.Read())
Console.WriteLine("-> {0}, {1}", dr["au_lname"], dr["au_fname"]);
dr.Close();
```

9.5 Tipuri nulabile

Pentru tipurile valoare este mandatorie stabiilirea unei valori; o variabilă de tip valoare nu poate să reţină `null`. Altfel spus, codul următor va genera eroare de compilare

```
static void Main(string[] args)
{
   bool myBool = null;
   int myInt = null;
}
```

În contextul lucrului cu baze de date este perfect posibil ca rezulattul unei interogări să aducă null pentru un anumit câmp. Pentru a rezolva această incompatibilitate (tipuri cu valoare nenulă în C# care trebuie să poată lucra cu null-urile provenite din bază), s-au introdus tipurile nulabile. Acesta reprezintă un mecansim de extindere a tipurilor de tip valoare astfel încât să suporte şi valoarea de nul.

De exemplu, pentru a putea declara o variabilă de tip int care să poată avea şi valoare nulă se va scrie:

```
int? intNulabil = null;
```

```
i=3;
i=null;
```

Nu vom putea defini ca nulabile tipurile referinţă, deoarece acestea suportă implicti null:

```
//eroare de compilare
string? s = null
```

Lucrul cu tipurile nulabile se face exact ca şi cu tipurile valoare:

```
class DatabaseReader
{
  //campuri nulabile
  public int? numbericValue;
  public bool? boolValue = true;

  public int? GetIntFromDatabase()
  { return numbericValue; }

  public bool? GetBoolFromDatabase()
  { return boolValue; }
}
```

În contextul tipurilor nulabile s-a introdus operatorul ?? care permite asignarea unei valori pentru o variabilă de tip nulabil dacă valoarea returnată este nulă:

```
static void Main(string[] args)
{
  DatabaseReader dr = new DatabaseReader();

  int? myData = dr.GetIntFromDatabase() ?? 100;
  Console.WriteLine("Value of myData: {0}", myData);
}
```

Curs 10

LINQ (I)

10.1 Generalităţi

Language Integrated Query (LINQ, pronunţat precum "link") permite interogarea unor colecţii de date folosind o sintaxă integrată în platforma .NET. Prin intermediul unor operatori se pot interoga colecţii de forma: vectori, colecţii, clase enumerabile, documente XML, baze de date relaţionale. Datele rezultate sunt văzute ca obiecte; are loc o mapare (asociere, traducere) a unor date neobiectuale într–un format uşor de folosit în limbajele obiectuale din cadrul plaformei. Trebuie menţionat că sintaxa este unitară, independent de natura sursei de date.

Listing 10.1: Interogare LINQ

```
var query = from e in employees
where e.id == 1
select e.name
```

sau:

Listing 10.2: Altă interogare LINQ

```
var results = from product in productsCollection
              where product.UnitPrice < 100
              select new {product.ProductName, product.UnitPrice};

foreach (var result in results)
{
        Console.WriteLine(result);
}
```

Este introdusă interfaţa `IQueryable<T>`, care permite implementarea unor furnizori de date în modul specific sursei de date considerate. Expresia

folosită pentru interogare este tradusă într-un arbore de expresie. Dacă colecţia implementează `IEnumerable<T>`, atunci se foloseşte motorul de execuţie LINQ local, integrat în platformă; dacă colecţia implementează `IQueryable<T>`, atunci se foloseşte implementarea bazată pe arborele de expresie; această implementare este dată de către furnizoare de LINQ. Furnizoarele de LINQ sunt scrise în mod specific fiecărei surse de date, dar datorită respectării unor interfeţe specificate, detaliile de implementare sunt irelevante pentru cel care foloseşte cod LINQ în interogare.

LINQ a fost introdus în versiunea 3.5 a lui .NET Framework. Constă într–un set de unelte care sunt folosite pentru lucrul cu date şi extensii aduse limbajului. Este alcătuit din:

- LINQ to Objects – se aduc date din colecţii care implementează `IEnumerable<T>`; datele interogate sunt deja în memoria procesului;
- LINQ to XML – converteşte documentele XML într–o colecţie de obiecte de tip `XElement`;
- LINQ to SQL – permite convertirea interogărilor LINQ în comenzi SQL;
- LINQ to DataSets – spre deosebire de LINQ to SQL care a venit iniţial doar cu suport pentru SQL Server, LINQ to DataSets foloseşte ADO.NET pentru comunicarea cu baze de date;
- LINQ to Entities – soluţie Object/Relational Mapping de la Microsoft ce permite utilizarea de Entities – introduse în ADO.NET 3.0 – pentru a specifica declarativ structura obiectelor ce modelează domeniul şi foloseşte LINQ pentru interogare.

La ora actuală există următorii furnizori de date LINQ:

1. LINQ to MySQL, PostgreSQL, Oracle, Ingres, SQLite şi Microsoft SQL Server
2. LINQ to CSV
3. LINQ to Google
4. LINQ to NHibernate
5. LINQ to System Search

Se lucrează la PLINQ, Parallel LINQ, un motor ce foloseşte paralelizarea de cod pentru executare mai rapidă a interogărilor, în cazul unui sistem multi-nucleu sau multiprocesor.

10.2 Motivaţie

Vom prezenta două motive pentru care LINQ este util:

- codul stufos, neproductiv utilizat pentru accesarea în modul clasic a datelor;
- nepotrivirea paradigmelor obiectual–relaţionale.

10.2.1 Codul clasic ADO.NET

Pentru accesarea datelor dintr-o bază de date relaţională, folosind ADO.NET se scrie de regulă un cod de forma:

Listing 10.3: Interogare clasică folosind ADO.NET

```
using (SqlConnection connection = new SqlConnection("..."))
{
 connection.Open();
 SqlCommand command = connection.CreateCommand();
 command.CommandText =
   @"SELECT Name, Country
     FROM Customers
     WHERE City = @City";
 command.Parameters.AddWithValue("@City", "Paris");
 using (SqlDataReader reader = command.ExecuteReader())
 {
   while (reader.Read())
   {
     string name = reader.GetString(0);
     string country = reader.GetString(1);
     ...
   }
 }
}
```

Se remarcă următoarele:

- cantitatea de cod scrisă; codul de sus este unul des folosit, scrierea lui în repetate rânduri este contraproductivă;
- interogările sunt exprimate prin intermediul unei fraze scrise între ghilimele, ca şir de caractere, deci automat nu se poate verifica prin compilarea corectitudinea codului SQL conţinut; este adevărat, însă, că se pot folosi aici proceduri stocate care sunt compilate deja pe server;

- slaba tipizare a parametrilor: dacă tipul acestora nu coincide cu ce se află în baza de date? dacă numărul de parametri este incorect (asta se semnalează numai la rulare, ca eroare; de preferat ar fi fost să se depisteze acest lucru la compilare);

- de cele mai multe ori trebui folosit un dialect de SQL specific producătorului serverului de baze de date; codul SQL nu este portabil; totodată: mixarea de limbaje – C# şi SQL – face codul greu de urmărit.

O expresie LINQ care care demonstrează depăşirea acestor probleme este:

Listing 10.4: Cod LINQ

```
from customer in customers
where customer.Name.StartsWith("A") &&
    customer.Orders.Count > 10
orderby customer.Name
select new { customer.Name, customer.Orders }
```

10.2.2 Nepotrivirea de paradigme

Paradigma este "o constructie mentală larg acceptată, care oferă unei comunităţi sau unei societăţi pe perioadă îndelungată o bază pentru crearea unei identităţi de sine (a activităţii de cercetare de exemplu) si astfel pentru rezolvarea unor probleme sau sarcini"[1].

Există o diferenţă sesizabilă între programarea orientată pe obiecte, utilizată în cadrul limbajelor folosite pentru implementarea aplicaţiilor şi modul de stocare şi reprezentare a datelor: XML sau baze de date relaţionale. Translatarea unui graf de obiecte din reprezentarea obiectuală într–o altă reprezentare este greoaie: programatorul trebuie să înţeleagă şi particularităţile structurilor de date folosite pentru persistarea lor, pe lângă cunoaşterea limbajului în care lucrează.

Problema pe care LINQ o abordează este "rezolvarea" următoarelor inegalităţi:

- "Data != Objects"

- "Relational data != Objects"

- "XML data != Objects"

- "XML data != Relational data"

[1] Wikipedia

Toate aceste nepotriviri necesită efort de adaptare din partea programatorului. Modelarea obiectual–relaţională este problema cea mai des întâlnită, cu următoarele aspecte:

1. tipurile de date folosite de către modelele relaţionale şi modelele obiectuale nu sunt aceleaşi; de exemplu, multitudinea de tipuri şir de caractere folosite în specificarea coloanelor, în timp ce în .NET există doar tipul `String`;

2. modelele relaţionale folosesc normalizarea (pentru eliminarea redundanţei şi a anomaliilor de inserare, ştergere, modificare), în timp ce în modelarea obiectuală nu trebuie să treacă prin aşa ceva; în schimb, modelarea obiectuală foloseşte agregarea sau moştenirea, mecanisme care nu îşi au un echivalent direct în modelarea relaţională

3. modele de programare diferite: pentru SQL se foloseşte un limbaj declarativ care specifică *ce* se prelucrează, în timp ce limbajele de programare folosite sunt de regulă imperative – arată *cum* se face prelucrarea

4. încapsulare – ascunderea detaliilor şi legarea laolaltă datelor cu metodele care prelucrează datele;

Toate aceste probleme se manifestă începând cu maparea între obiecte şi datele persistate. Aceeaşi problemă apare dacă ne referim la XML, care favorizează un model ierarhic, semistructurat. Programatorul trebuie să scrie mereu un cod care să faciliteze legarea acestor universuri diferite. LINQ vine cu o propunere de rezolvare.

10.3 LINQ to Objects: exemplificare

LINQ to Objects este folosit pentru interogarea datelor care se află deja în memorie. Un prim exemplu este:

```
using System;
using System.Linq;
static class HelloWorld
{
 static void Main()
 {
  string[] words =
  { "hello", "wonderful", "linq", "beautiful", "world" };
```

```
   var shortWords =
    from word in words
    where word.Length <= 5
    select word;
   foreach (var word in shortWords)
   Console.WriteLine(word);
  }
}
```

Un exemplu mai complex este:

Listing 10.5: LINQ peste colecţie generică, folosind expresie LINQ

```
List<Person> people = new List<Person> {
  new Person() { ID = 1,
    IDRole = 1,
    LastName = "Anderson",
    FirstName = "Brad"
  },
  new Person() {
   ID = 2,
   IDRole = 2,
   LastName = "Gray",
   FirstName = "Tom"
  }
};
var query =
 from p in people
 where p.ID == 1
 select new { p.FirstName, p.LastName };
```

Interogarea din final poate fi scrisă şi altfel:

Listing 10.6: LINQ peste colecţie generică, folosind apeluri de metode

```
var query = people
    .Where(p => p.ID == 1)
    .Select(p => new { p.FirstName, p.LastName } );
```

10.4 Mecanisme utilizate de LINQ

Pentru a putea fi folosit, LINQ a necesitat extinderea limbajelor din cadrul platformei.

10.4.1 Inferenţa tipului

Considerăm metoda:

```
void f()
{
  int x = 3;
  MyClass y = new MyClass();
  bool z = true;
  ....
}
```

Pentru fiecare din declaraţiile cu iniţializare de mai sus se poate spune că valoarea asociată dă suficientă informaţie despre tipul de date corespunzător variabilelor. Tocmai din acest motiv în C# 3.0 se poate scrie astfel:

```
void f()
{
  var x = 3;
  var y = new MyClass();
  var z = true;
  ....
}
```

`var` nu reprezintă un nou tip de date, ci pur şi simplu arată că la compilare se poate deduce tipul actual al variabilelor locale respective. Ca atare, `var` este de fapt o scurtătură, prin care se obţine ceea ce s-a scris prima oară. Dacă după declarare şi iniţializare se scrie:

```
x = false;
```

compilatorul semnalează eroare, deoarece s-a făcut deja inferarea tipului de dată pentru `x`, şi anume `int`, iar `false` nu este compatibil cu `int`. Limbajul rămâne deci puternic tipizat, fiecare variabilă având un tip asignat.

Folosirea acestui nou cuvânt cheie se supune condiţiilor:

- se poate folosi doar pentru variabile locale
- se poate folosi atunci când compilatorul poate să infereze tipul de dată asociat variabilei; acest lucru se întâmplă conform situaţiilor de mai sus, sau pentru o iterare de forma:

  ```
  int[] sir = {1, 2, 3};
  foreach(var i in sir)
  ```

```
{
  ...
}
```

- odată ce compilatorul determină tipul de date, acesta nu mai poate fi schimbat.

Mecanismul nu reprezintă la prima vedere un mare pas, dar este cu adevărat util atunci când se lucrează cu alte mecanisme din C# 3.0: tipuri anonime şi programarea bazată pe LINQ.

10.4.2 Tipuri anonime

Acest mecanism permite declararea unor variabile de un tip care nu este definit aprioric. Se omite declararea numelui de clasă şi a componenţei acesteia. Exemplu:

```
var p = new {FirstName="Rafael", Age=25};
```

Pentru proprietăţile care sunt pomenite în expresia de iniţializare se face deducerea tipului în mod automat (pe baza aceluiaşi mecanism de la variabile anonime). Proprietăţile sunt publice şi read-only.

10.4.3 Metode parţiale

Metodele parţiale reprezintă metodele care sunt declarate în mai multe părţi ale unei clase. Clasa conţinătoare poate să fie sau nu parţială. Cele două părţi sunt una în care se defineşte metoda parţială (cu tip de retur şi parametri, dar fără corp) şi alta în care se implementează (cu corp complet).

Exemplu:

```
//fisierul MyClass1.cs
partial class MyClass
{
    //definitie de metoda
    partial void f(int x);
}

//fisierul MyClass2.cs
partial class MyClass
{
    //implementare de metoda
```

```
    partial void f(int x)
    {
        Console.WriteLine(x.ToString());
    }
}
```

Compilatorul va pune la un loc cele două declaraţii de metode parţiale şi va rezulta o metodă "întreagă". Regulile care trebuie respectate pentru crearea de metode parţiale sunt:

1. metoda pentru care se foloseşte implementare parţială trebuie să returneze `void`;
2. parametrii nu pot fi de tip `output`;
3. metoda nu poate avea specificatori de acces; ea este privată
4. metoda trebuie să fie declarată atât la definire cât şi la implementare ca fiind parţială.

Adăgăm că o astfel de metodă parţială poate să apară într–o structură sau într–o clasă (declarate ca parţiale). Nu este obligatoriu ca ambele declaraţii să apară în părţi diferite ale clasei. Dacă o implementare de metodă parţială lipseşte, atunci orice apel la ea este ignorat.

10.4.4 Metode de extensie

Metodele de extensie permit scrierea de metode asociate cu clase, alte clase decât cele în care sunt definite.

Să considerăm o clasă, în cazul căreia fiecare obiect menţine un şir de numere. O posibilă definiţie ar fi:

```
class LotOfNumbers
{
    private int[] numbers;

    public LotOfNumbers(int[] numbers)
    {
        this.numbers = new int[numbers.Length];
        numbers.CopyTo(this.numbers, 0);
    }

    public int NumbersNo
```

```
    {
        get
        {
            if (numbers == null)
            {
                return 0;
            }
            else
            {
                return numbers.Length;
            }
        }
    }

    public int Sum()
    {
        if (numbers == null)
        {
            throw new Exception("No number inside");
        }
        int result = 0;
        foreach (int x in numbers)
        {
            result += x;
        }
        return result;
    }
}
```

Ne propunem să adăugăm o metodă la această clasă, care să returneze media elementelor şirului. Să presupunem că nu avem acces la sursa codului C#. Singura modalitate de a extinde clasa `LotOfNumbers` este ca să se scrie o metodă de extensie:

```
static class ExtendsLotOfNumbers
{
    public static double Average(this LotOfNumbers data)
    {
        return (double)data.Sum() / data.NumbersNo;
    }
}
```

Utilizarea metodei de extensie se face cu:

```
class Program
{
    static void Main()
    {
        LotOfNumbers numbers = new LotOfNumbers(new int[]{1, 2, 3});
        Console.WriteLine(numbers.Average().ToString());
    }
}
```

Am reusit astfel să "strecurăm" o metodă în interirorul unei clase al cărei cod sursă nu este accesibil. Putem utiliza acest mecanism dacă:

1. clasa în care se face implementarea metodei de extensie este statică;
2. metoda care implementează extensia este statică
3. metoda de extensie are primul parametru de tipul clasei pentru care se face extinderea, iar tipul parametrului formal este prefixat cu `this`.

Este posibil ca o metodă de extensie să aibe mai mult de un parametru.

10.4.5 Expresii lambda

Aceste expresii simplifică scrierea delegaţilor şi a metodelor anonime. Într–un exemplu anterior s–a folosit:

```
Where(p => p.ID == 1)
```

care s–ar citi: "obiectul `p` produce expresia logică p.ID egal cu 1". Ar putea fi rescrisă echivalent, astfel:

Listing 10.7: Expresie lambda rescrisă cu delegaţi

```
Func<Person, bool> filter = delegate(Person p) { return p.ID == 1; };
var query = people
  .Where(filter)
  .Select(p => new { p.FirstName, p.LastName } );
```

10.5 Operatori LINQ

Tabelul 10.1: Operatori LINQ.

Operaţie	Operator	Descriere
Aggregate	`Aggregate`	Aplică o funcţie peste o secvenţă
	`Average`	Calculează media peste o secvenţă
	`Count/LongCount`	calculează numărul de elemente dintr-o secvenţă
	`Max`, `Min`, `Sum`	Calculează maximul, minimul, suma elementelor dintr-o secvenţă
Concatenare	`Concat`	Concatenează elementele a două secvenţe
Conversie	`AsEnumerable`	Converteşte o secvenţă la un `IEnumerable<T>`
	`AsQueryable`	Converteşte o secvenţă la un `IQueryable<T>`
	`Cast`	converteşte un element al unei secvenţe la un tip specificat
	`OfType`	Filtrează elementele unei secvenţe, returnându–le doar pe cele care au un tip specificat
	`ToArray`	transformă în vector
	`ToDictionary`	transformă în dicţionar
	`ToList`	transformă în colecţie
	`ToLookup`	creează un obiect de tip `Lookup<K, T>` dintr-o secvenţă
	`ToSequence`	returnează argumentul transformat într–un `IEnumerable<T>`
Obţinere de element	`DefaultIfEmpty`	dă un element implicit dacă secvenţa este goală
	`ElementAt`	returnează elementul de la poziţia specificată
	`ElementAtOrDefault`	returnează elementul de la poziţia specificată sau o valoare implicită, dacă la poziţia specificată nu se află nimic
	`First`, `Last`	primul, respectiv ultimul element dintr–o secvenţă
	`FirstOrDefault` `LastOrDefault`	ca mai sus, dar cu returnare de valoare implicită dacă primul, respectiv ultimul elem din colecţie nu este diponibil
	`Single`	returnează elementul din colecţie,

Tabelul 10.1 (continuare)

Operaţie	Operator	Descriere
		presupusă a fi format dintr–un singur element
	`SingleOrDefault`	ca mai sus, sau element implicit dacă elementul singular nu este găsit în secvenţă
Egalitate	`SequenceEqual`	verifică dacă două secvenţe sunt egale
Generare	`Empty`	returnează o secvenţă goală de tipul specificat
	`Range`	Generează o secvenţă de numere aflate între două capete specificate
	`Repeat`	Generează o secvenţă prin repetarea de un număr de ori specificat a unei valori specificate
Grupare	`GroupBy`	Grupează elementele unei secvenţe
Joncţiune	`GroupJoin`	O jonţiune grupată a două secvenţe pe baza unor chei care se potrivesc
	`Join`	Jonţiune interioară a două secvenţe
Sortare	`OrderBy`	Ordonează elementele unei secvenţe pe baza unora sau a mai multo chei
	`OrderByDescending`	Ca mai sus, dar cu sortare descrescătoare
	`Reverse`	inversează ordinea elementelor dintr-o secvenţă
	`ThenBy`, `ThenByDescending`	pentru specificarea de chei suplimentare de sortare
Partiţionare	`Skip`	Produce elementele unei secvenţe care se află după o anumită poziţie
	`SkipWhile`	Sare peste elementele unei secvenţe care îndeplinesc o anumită condiţie
	`Take`	Ia primele elemente dintr-o secvenţă
	`TakeWhile`	Ia elementele de la începutul unei secvenţe, atâta timp cât ele respectă o anumită condiţie
Proieţie	`Select`	defineşte elementele care se iau în secvenţă
	`SelectMany`	Preia elemente dintr-o secvenţă conţinând secvenţe
Cuantificatori	`All`	Verifică dacă toate elementele unei secvenţe satisfac o condiţie dată
	`Any`	Verifică dacă vreun elementl al unei secvenţe satisface o condiţie dată
	`Contains`	Verifică dacă o secvenţă conţine un element

Tabelul 10.1 (continu

Operaţie	Operator	Descriere
Restricţie	`Where`	Filtrează o secvenţă pe baza unei condiţii
Mulţime	`Distinct`	Returnează elementele distincte dintr–o colecţie
	`Except`	Efectuează diferenţa a două secvenţe
	`Intersect`	Returnează intersecţia a două mulţimi
	`Union`	Produce reuniunea a două secvenţe

10.6 LINQ to Objects

Exemplele de mai jos sunt preluate din [8] şi sunt folosite pentru exemplificarea codului ce foloseşte LINQ. Se pleacă de la clase `Person`, `Role` şi `Salary`, definite ca mai jos:

Listing 10.8: Clase pentru exemplificarea LINQ-ului

```
class Person
{
 public int ID
 {
  get;
  set;
 }

 public int IDRole
 {
  get;
  set;
 }

 public string LastName
 {
  get;
  set;
 }

 public string FirstName
 {
  get;
```

```
  set;
 }
}

class Role
{
 public int ID
 {
  get;
  set;
 }

 public string RoleDescription
 {
  get;
  set;
 }
}

class Salary
{
 public int IDPerson
 {
  get;
  set;
 }

 public int Year
 {
  get;
  set;
 }

 public double SalaryYear
 {
  get;
  set;
 }
}
```

10.6.1 Filtarea cu `Where`

Pentru operatorul `Where` s–au definit două metode de extensie:

```
public static IEnumerable<T> Where<T>(
 this IEnumerable<T> source, Func<T, bool> predicate);

public static IEnumerable<T> Where<T>(
 this IEnumerable<T> source, Func<T, int, bool> predicate);
```

Prima formă foloseşte un predicat (condiţie) care pentru un obiect de tipul `T` returnează un boolean, iar în al doilea caz se foloseşte la condiţie obiectul şi indicele său în secvenţă.

Să presupunem că avem o colecţie de obiecte de tip `Person` construită astfel:

```
List<Person> people = new List<Person> {
 new Person {
  ID = 1,
  IDRole = 1,
  LastName = "Anderson",
  FirstName = "Brad"
 },
 new Person {
  ID = 2,
  IDRole = 2,
  LastName = "Gray",
  FirstName = "Tom"
 },
 new Person {
  ID = 3,
  IDRole = 2,
  LastName = "Grant",
  FirstName = "Mary"
 },
 new Person {
  ID = 4,
  IDRole = 3,
  LastName = "Cops",
  FirstName = "Gary"
 }
};
```

Obţinerea obiectelor de tip `Person` care au prenumele "Brad" se face cu *expresia LINQ*:

```
var query = from p in people
 where p.FirstName == "Brad"
 select p;
```

Elementele care sunt aduse de către interogarea anterioară pot fi iterate cu:

```
foreach (Person x in query)
{
  Console.WriteLine("{0}, {1}", x.FirstName, x.LastName);
}
```

Pentru expresia de interogare de mai sus se poate scrie echivalent:

```
var query = people.Where(p => p.FirstName == "Brad");
```

adică o variantă în care se folosesc metodele de extensie.

Dacă se doreşte folosirea în cadrul condiţiei de filtrare a poziţiei elementului curent în lista `people`, se poate scrie:

```
var query = people.Where((p, index) => p.IDRole == 1
  && index % 2 == 0);
```

şi utilizarea metodei de extensie este singura posibilitate de a referi indicele d epoziţie.

10.6.2 Operatorul de proiecţie

Pentru a determina ce conţine fiecare element care compune o secvenţă, se foloseşte operatorul `Select`, definit de metodele de extensie:

```
public static IEnumerable<S> Select<T, S>(
   this IEnumerable<T> source, Func<T, S> selector);

public static IEnumerable<S> Select<T, S>(
   this IEnumerable<T> source, Func<T, int, S> selector);
```

Selectarea se poate face cu:

```
var query = from p in people
   select p.FirstName;
```

sau:

```
var query = people.Select(p => p.FirstName);
```

Se poate de asemenea ca la fiecare element selectat să se producă un obiect de un tip nou, chiar tip anonim:

```
var query = people
 .Select(
  (p,index) => new {
    Position=index,
    p.FirstName,
    p.LastName
  }
  );
```

10.6.3 Operatorul `SelectMany`

Are metodele de extensie:

```
public static IEnumerable<S> SelectMany<T, S>(
 this IEnumerable<T> source,
 Func<T, IEnumerable<S>> selector);

public static IEnumerable<S> SelectMany<T, S>(
 this IEnumerable<T> source,
 Func<T, int, IEnumerable<S>> selector);
```

Să presupunem că specificăm şi nişte obiecte rol:

```
List<Role> roles = new List<Role> {
 new Role { ID = 1, RoleDescription = "Manager" },
 new Role { ID = 2, RoleDescription = "Developer" }};
```

Dacă dorim rolul persoanei având id-ul 1, atunci putem scrie:

```
var query = from p in people
 where p.ID == 1
 from r in roles
 where r.ID == p.IDRole
 select new
 {
  p.FirstName,
  p.LastName,
  r.RoleDescription
 };
```

Echivalent, se poate folosi metoda `SelectMany`:

```
var query = people
 .Where(p => p.ID > 1)
 .SelectMany(p => roles
```

```
            .Where(r => r.ID == p.RoleID)
            .Select(r => new {
             p.FirstName,
             p.LastName,
             r.RoleDescription
            }
           )
        );
```

`SelectMany` permite gestiunea unei alte secvenţe elemente; dacă s-ar folosi `Select` în loc de `SelectMany`, atunci s–ar obţine o secvenţă de elemente de tip `IEnumerable<T>`.

10.6.4 Joncţiuni

Operatorul `Join` poate fi folosit pentru a aduce date din două colecţii, date care sunt puse în corespondenţă pe baza unei chei. Se bazează pe metoda de extensie:

```
public static IEnumerable<V> Join<T, U, K, V>(
 this IEnumerable<T> outer,
 IEnumerable<U> inner,
 Func<T, K> outerKeySelector,
 Func<U, K> innerKeySelector,
 Func<T, U, V> resultSelector);
```

De exemplu, pentru a aduce lista numelor, prenumelor şi a descrierilor de roluri pentru colecţiile `persons` şi `roles` putem folosi:

```
var query = from p in people
 join r in roles on p.IDRole equals r.ID
 select new {
  p.FirstName,
  p.LastName,
  r.RoleDescription
 };
```

Folosind operatori (metode) LINQ, se poate scrie:

```
query = people.Join(roles, p => p.IDRole, role => role.ID,
  (p, r) => new {
   p.FirstName,
   p.LastName,
   r.RoleDescription
```

```
    }
);
```

10.6.5 Grupare

Gruparea datelor se face cu `GroupBy`; se grupează elementele unei secvenţe pe baza unui selector.

Exemplu: plecând de la

```
var query = from m in typeof(int).GetMethods()
  select m.Name;
```

Dacă se afişează elementele din colecţia `query`, se poate observa că metoda `ToString` este afişată de 4 ori, `Equals` de couă ori etc. Gruparea acestora[2] se poate face astfel:

```
var q = from m in typeof(int).GetMethods()
 group m by m.Name into gb
 select new {Name = gb.Key};
```

Dacă dorim să afişăm şi o valoare agregată la nivel de grup, atunci putem specifica suplimentar în operatorul de selecţie:

```
var q = from m in typeof(int).GetMethods()
 group m by m.Name into gb
 select new {Name = gb.Key, Overloads = gb.Count()};
```

Pentru grupare se poate specifica şi un comparator (implementare de interfaţă `IEqualityComparer`) care să spună care este criteriul de determinare a elementelor egale ce formează un grup.

10.6.6 Ordonare

Sunt folosiţi operatorii: `OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending` şi `Reverse`.

Operatorii `OrderBy` şi `OrderByDescending` produc sortarea crescătoare, respectiv descrescătoare a unor secvenţe, pe baza unei chei de sortare. Exemplu:

```
var q = from m in typeof(int).GetMethods()
        orderby m.Name
        select new { Name = m.Name };
```

Folosind metode se poate scrie:

[2] A nu se confunda exemplul cu eliminarea duplicatelor.

```
q = typeof(int).
  GetMethods().
  OrderBy(method => method.Name).
  Select(method => new { Name = method.Name });
```

Pentru sortarea descrescătoare expresia LINQ este:

```
var q = from m in typeof(int).GetMethods()
        orderby m.Name descending
        select new { Name = m.Name };
```

sau prin operatori LINQ:

```
q = typeof(int).
  GetMethods().
  OrderByDescending(method => method.Name).
  Select(method => new { Name = method.Name });
```

Dacă se doreşte specificarea mai multor chei după care să se facă sortarea în ordine lexicografică, atunci se poate specifica prin `ThenBy` şi `ThenByDescending` – apelate ca metode – care sunt criteriile suplimentare de sortare. Dacă se foloseşte expresie LINQ, atunci se poate scrie mai simplu:

```
var query = from p in people
 orderby p.FirstName, p.LastName
 select p;
```

Exprimarea echivalentă cu operatori LINQ este:

```
var query = people.OrderBy(p => p.FirstName)
      .ThenBy(p => p.LastName);
```

Pentru personalizarea sortării, se poate folosi o implementare de interfaţă `IComparer<T>`.

Inversarea ordinii elementelor dintr–o secvenţă se face cu `Reverse`, apelată ca metodă:

```
var q = (from m in typeof(int).GetMethods()
                    select new { Name = m.Name }).Reverse();
```

sau:

```
var q = typeof(int)
     .GetMethods()
     .Select(method => new { Name = method.Name })
     .Reverse();
```

10.6.7 Agregare

Există operatorii de agregare: `Count`, `LongCount`, `Sum`, `Min`, `Max`, `Average`, `Aggregate`.

Metoda `Count()` returnează un întreg, iar `LongCount()` un long (intreg pe 64 de biţi, cu semn) reprezentând numărul de elemente din secvenţă.

Metoda `Sum` calculează suma unor valori numerice dintr–o secvenţă. Metodele sunt:

```
public static Numeric Sum(
 this IEnumerable<Numeric> source);

public static Numeric Sum<T>(
 this IEnumerable<T> source,
 Func<T, Numeric> selector);
```

unde `Numeric` se referă la un tip de date de forma: `int, int?, long, long?, double, double?, decimal`, sau `decimal?`.

Exemplu:

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var query = numbers.Sum();
Console.WriteLine(query.ToString());
```

Dacă dorim să determinăm care este suma salariilor anuale primite de către fiecare salariat în parte, atunci putem forma prima oară grupuri bazate pe numele de familie al salariatului şi apoi se poate face suma salariilor de-a lungul anilor. Presupunem că salariile sunt date astfel:

```
List<Salary> salaries = new List<Salary> {
   new Salary {
    IDPerson = 1,
    Year = 2004,
    SalaryYear = 10000.00 },
   new Salary {
    IDPerson = 1,
    Year = 2005,
    SalaryYear = 15000.00 }
   };

var query =
 from p in people
 join s in salaries on p.ID equals s.IDPerson
 select new
```

```
 {p.FirstName,
  p.LastName,
  s.SalaryYear
 };
```

Apoi:

```
var querySum = from q in query
 group q by q.LastName into gp
 select new {
         LastName = gp.Key,
         TotalSalary = gp.Sum(q => q.SalaryYear)
        };
```

Pentru operatorii Min, Max, Average, aceştia trebuie apelaţi pentru o colecţie de valori numerice.

Operatorul `Aggregate` permite definirea unei metode care să fie folosită pentru sumarizarea datelor. Declaraţia metodei este:

```
public static T Aggregate<T>(
 this IEnumerable<T> source,
 Func<T, T, T> func);

public static U Aggregate<T, U>(
 this IEnumerable<T> source,
 U seed,
 Func<U, T, U> func);
```

A doua metodă sepcifică prin intermediul valorii `seed` care este valoarea de început cu care se porneşte agregarea. Dacă nu se specifică nicio valoare pentru `seed`, atunci primul element din colecţie este luat drept `seed`.

Pentru înmulţirea valorilor cuprinse într–un tablou se poate proceda astfel:

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var query = numbers.Aggregate((a,b) => a * b);
```

Valoarea afişată este 9! = 362880. Specificarea unei valori pentru `seed` se face (exemplu):

```
int[] numbers = { 9, 3, 5, 4, 2, 6, 7, 1, 8 };
var query = numbers.Aggregate(5, (a,b) => ( (a < b) ? (a * b) : a));
```

10.6.8 Partiţionare

Se folosesc pentru extragerea unei anumite părţi dintr-o secvenţă. Operatorii sunt: `Take`, `Skip`, `TakeWhile`, `SkipWhile`. Vom exemplifica doar o parte din

aceştia:

```
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
var query = numbers.Take(5);//secventa 1, 2, 3, 4, 5
var query2 = numbers.Skip(5);//secventa 6, 7, 8, 9
var query3 = numbers.TakeWhile((n, index) => n + index < 4);//1, 2
var query4 = numbers.SkipWhile((n, index) => n + index < 4);//3, 4,
```

10.6.9 Concatenarea

Se face cu `Concat`, care preia două secvenţe şi produce o a treia, reprezentând concatenarea lor, fără eliminarea duplicatelor:

```
int[] x = { 1, 2, 3 };
int[] y = { 3, 4, 5 };
var concat = x.Concat(y);//concat = 1, 2, 3, 3, 4, 5
```

10.6.10 Referirea de elemente din secvenţe

Se pot folosi: `First`, `FirstOrDefault`, `Last`, `LastOrDefault`, `Single`, `SingleOrDefault`, `ElementAt`, `ElementAtOrDefault` şi `DefaultIfEmpty`.

Pentru `First` şi derivaţii lui există formele:

```
public static T First<T>(
 this IEnumerable<T> source);

public static T First<T>(
 this IEnumerable<T> source,
 Func<T, bool> predicate);

public static T FirstOrDefault<T>(
 this IEnumerable<T> source);

public static T FirstOrDefault<T>(
 this IEnumerable<T> source,
 Func<T, bool> predicate);
```

Dacă nu se specifică predicatul, atunci se va returna primul element din secvenţă, altfel primul element din secvenţă care satisface condiţia exprimată prin predicat.

```
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
var query = numbers.First();
query = numbers.First(n => n % 2 == 0);
```

Varianta cu `OrDefault` este utilă dacă se crede că predicatul poate să nu fie satisfăcut de niciun element al secvenţei. Se returnează valoarea implicită pentru tipul de date considerat: 0 pentru tip numeric, false pentru boolean, null pentru tip referinţă. Dacă se foloseşte varianta fără `OrDefault`, atunci e posibil ca aplicaţia să se oprească cu eroare – asta în cazul în care niciun element al colecţiei nu respectă condiţia.

Curs 11

LINQ (II): Linq to SQL

Linq to SQL vine cu o soluţie pentru accesarea datelor stocate într-un server de baze de date SQL Server 2000, 2005 sau 2008; se permite şi implementarea unor furnizori de Linq pentru alte servere de baze de date[1]. Linq to SQL defineşte o asociere între tabelele din baza de date şi clase C#, clase numite entităţi. Suplimentar, se pune la dispoziţie o clasă `DataContext` care mediază comunicarea între baza de date şi clasele entitate. `DataContext` menţine starea obiectelor, transformă dintr-un obiect într–o înregistrare şi invers.

11.1 Obţinerea unui context de date

Obţinerea obiectului de tip `DataContext` se poate face adăugând un obiect de tip "LINQ to SQL Classes", prin "Add new item" la nivel de aplicaţie, precum în figura 11.1. Va rezulta adăugarea unui fişier cu extensia dbml la aplicaţie; acesta conţine o clasă derivată din `DataContext`.

Prin adăugarea tabelelor din fereastra de Server Explorer se vor crea şi clasele entitate aferente tabelelor (figurile 11.2 şi 11.3).

Obţinerea unui obiect asociate tabelei `Person` se face prin:

```
PersonDataContext pdc = new PersonDataContext();
```

Trebuie avut în vedere faptul că clasa asociată contextului de date implementează interfaţa `IDisposable`, deci un asemenea obiect poate şi ar trebui să fie disponibilizat de îndată ce nu mai este nevoie de serviciile lui. Acest lucru se face fie prin instrucţiunea `using`, fie prin bloc `try-finally`:

```
using (PersonDataContext pdc = new PersonDataContext())
{
```

[1] Vezi de exemplu : furnizor pentru MySql, Oracle, PostgreSQL.

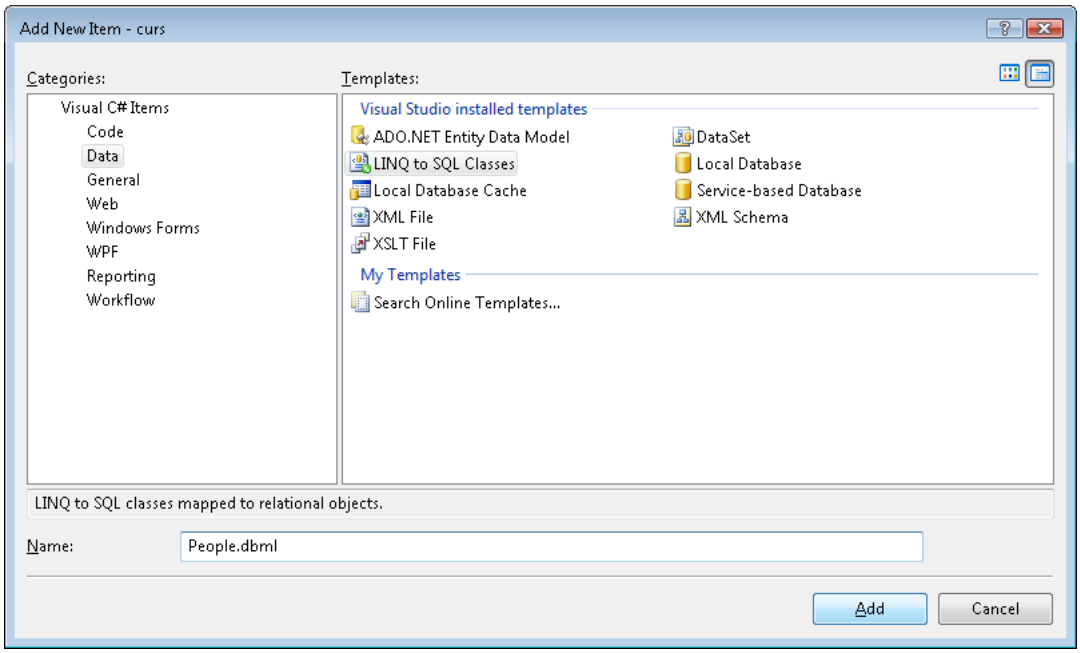

Figura 11.1: Adăgarea unui fişier DBML la aplicaţie

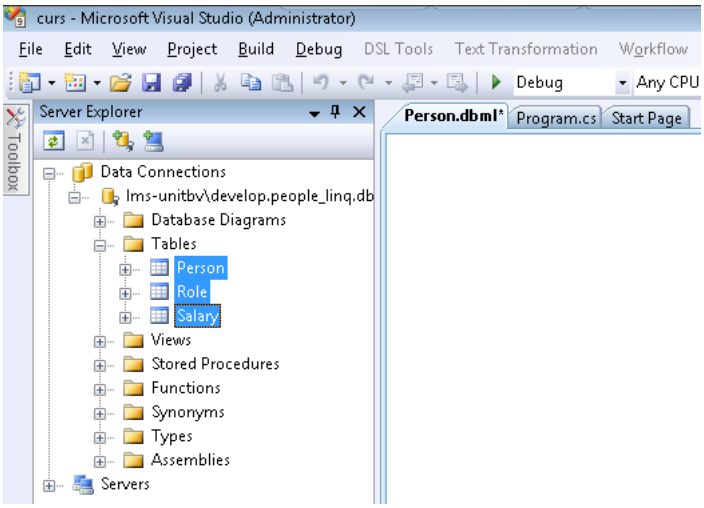

Figura 11.2: Selectarea tabelor pe baza cărora se va construi fişierul DBML

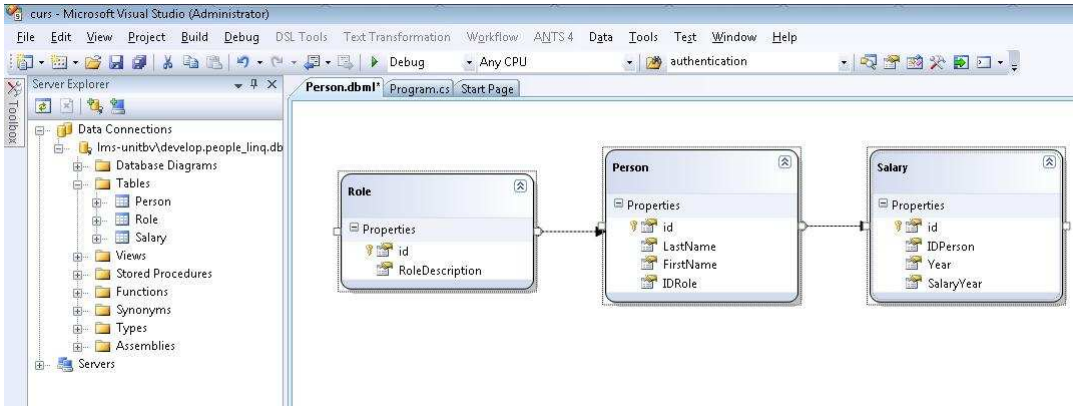

Figura 11.3: Clasele C# rezultate după adăugarea de tabele pe suprafaţa fişierului DBML

```
    Table<Person> personTable = pdc.GetTable<Person>();
}
//sau
PersonDataContext pdc = new PersonDataContext();
try
{
...
}
finally
{
    pdc.Dispose();
}
```

Obţinerea unui obiect asociat tabelei People se face prin:

```
Table<Person> personTable = pdc.GetTable<Person>();
```

iar obţinerea de obiecte de tip `Person`, câte unul asociat fiecărei înregistrări se face cu:

```
var allPersons = from p in personTable
                 select p;
```

Mai mult, orice interogare efectuată cu expresii Linq sau operatori Linq poate fi efectuată asupra tabelelor. De exemplu, pentru o joncţiune între tabelele `Person` şi `Salary` putem scrie:

```
var personsWithSalaries = from p in pdc.Persons
                          join s in pdc.Salaries
                          on p.id equals s.IDPerson
```

```
                                    select new { p.FirstName, p.LastName,
                                    s.Year, s.SalaryYear };

foreach (var iter in personsWithSalaries)
{
    Console.WriteLine("FirstName: {0} LastName: {1} Year: {2}
    SalaryYear: {3}", iter.FirstName, iter.LastName,
    iter.Year.ToString(), iter.SalaryYear.ToString());
}
```

Pentru a vizualiza interogarea SQL care este formulată spre a se trimite către server, se poate scrie:

```
pdc.Log = Console.Out;
```

O altă variantă este folosirea obiectului de context de date în conjuncţie cu metoda `GetCommand` care primeşte interogarea pentru care se vrea vizualizarea comenzii SQL:

```
Console.WriteLine(pdc.GetCommand
  (personsWithSalaries).CommandText);
```

11.2 Adăugarea, modificarea şi ştergerea de înregistrări în tabelă

11.2.1 Adăugarea

Pentru adăugarea unei noi înregistrări, se poate crea un obiect de tipul entitate - clasă furnizată automat de către DBML - iar apoi apelarea metodei `InsertOnSubmit` pentru tabela în care se dorecşte inserarea. Adăugarea la setul de înregistrări se face numai când se apelează metoda `SubmitChanges` pentru contextul de date:

```
using (PersonDataContext pdc = new PersonDataContext())
{
    Person p = new Person { FirstName = "inserted",
      LastName = "from code", IDRole = 1 };
    pdc.Persons.InsertOnSubmit(p);
    pdc.SubmitChanges();
}
```

Dacă pentru obiectul `p` se stabileşte rolul nu prin asigmare directă a valorii cheii străine, ci prin crearea unui obiect nou de tip `Role`, atunci se face

inserare automată a acestui nou obiect de rol şi apoi inserarea înregistrării de persoană, cu referinţă către rolul creat:

```
using (PersonDataContext pdc = new PersonDataContext())
{
  pdc.Log = Console.Out;
  Person p = new Person { FirstName = "inserted", LastName = "from code",
    Role = new Role { RoleDescription = "tester" } };
  pdc.Persons.InsertOnSubmit(p);
  pdc.SubmitChanges();
}
```

11.2.2 Modificarea unei înregistrări

Modificarea unei înregistrări se poate face prin obţinerea obiectului care va fi modificat, se modifică proprietăţile lui şi în final apelul metodei `SubmitChanges` pentru contextul de date:

```
using (PersonDataContext pdc = new PersonDataContext())
{
  pdc.Log = Console.Out;
  Person person = pdc.Persons.Where(p => p.id == 1).Single();
  person.LastName = "Modified";
  pdc.SubmitChanges();
}
```

Menţionăm doar că se poate face modificarea unui obiect care a fost adus într–un context deschis şi închis anterior, dar cu reataşare la context.

11.2.3 Ştergerea unei înregistrări

Printr–o manevră asemănătoare cu cea de mai sus, se preia înregistrarea care trebuie ştearsă şi apoi pentru obiectul rezultat se face ştergerea via metoda `DeleteOnSubmit` pentru obiectul reprezentând tabelul din care se şterge:

```
using (PersonDataContext pdc = new PersonDataContext())
{
  pdc.Log = Console.Out;
  Person person = pdc.Persons.Where(p => p.id == 9).
  DefaultIfEmpty().Single();
  if (person != null)
  {
```

```
        pdc.Persons.DeleteOnSubmit(person);
    }
    pdc.SubmitChanges();
}
```

11.3 Opţiuni de încărcare a datelor

Să presupunem că dorim să aducem lista de rolurim iar pentru fiecare rol persoanele care fac parte din el:

```
using (PersonDataContext pdc = new PersonDataContext())
{
    pdc.Log = Console.Out;
    var queryRoles = from role in pdc.Roles
                     select role;
    foreach (var role in queryRoles)
    {
        foreach (var person in role.Persons)
        {
            Console.WriteLine("person: FirstName {0}, LastName {1}",
             person.FirstName, person.LastName);
        }
    }
}
```

Se poate observa că la fiecare executare a ciclului `foreach` exterior se va apela câte o instrucţiune SQL care aduce persoanele care fac parte din rolul respectiv. Dată fiind abilitatea SQL-ului de a face joncţiuni, ne dorim să îmbunătăcţim acest ecomportament. Se poate obţine o încărcare completă a datelor prin utilizarea obiectului `DataLoadOptions`:

```
using (PersonDataContext pdc = new PersonDataContext())
{
    pdc.Log = Console.Out;
    DataLoadOptions dlo = new DataLoadOptions();
    dlo.LoadWith<Role>(role => role.Persons);
    pdc.LoadOptions = dlo;
    var queryRoles = from role in pdc.Roles
                     select role;
    foreach (var role in queryRoles)
    {
        foreach (var person in role.Persons)
```

```
    {
      Console.WriteLine("role: {0} person: FirstName {1},
        LastName {2}", role.RoleDescription,
        person.FirstName, person.LastName);
    }
  }
}
```

Curs 12

Atribute. Fire de execuţie

12.1 Atribute

O aplicaţie .NET conţine cod, date şi *metadate*. Metadata este o dată despre assembly (versiune, producator, etc), tipuri de date conţinute, etc stocate împreună cu codul compilat.

Atributele sunt folosite pentru a da o extra-informaţie compilatorului de .NET. Java foloseşte o combinaţie de semne /** şi @ pentru a include informaţie relativ la clase, metode, câmpuri sau parametri. Aceste comentarii însă nu vor fi incluse în bytecod-ul final, doar în eventuala documentaţie. Folosind atributele, această informaţie poate fi stocată în codul compilat şi reutilizată ulterior la runtime.

Unde ar fi utile aceste atribute? Un exemplu de utilizare a lor ar fi urmărirea bug–urilor care există într–un sistem sau urmărirea stadiului proiectului. De asemenea, anumite atribute predefinite sunt utilizate pentru a specifica dacă un tip este sau nu serializabil, care sunt porţiunile de cod scoase din circulaţie, informaţii despre versiunea assembly-ului, etc.

12.1.1 Generalităţi

Atributele sunt de două feluri: intrinseci (predefinite) şi definite de utilizator. Cele intrinseci sunt integrate în platforma .NET şi sunt recunoscute de CLR. Atributele definite de utilizator sunt create în funcţie de dorinţele acestuia.

Atributele se pot specifica pentru: assembly–uri, clase, constructori, delegaţi, enumerări, evenimente, câmpuri, interfeţe, metode, module, parametri, proprietăţi, valori de retur, structuri.

Ţinta unui atribut specifică cui anume i se va aplica un atribut anume. Tabelul 12.1 conţine ţintele posibile şi o scurtă descriere a lor.

Tabelul 12.1: Ţintele atributelor.

Ţintă	Descriere
All	Orice element
Assembly	Un assembly
ClassMembers	Orice membru al unei clase
Class	O clasă
Constructor	Un constructor
Delegate	Un delegat
Enum	O enumerare
Event	Un eveniment
Field	Un câmp
Interface	O interfaţă
Method	O metodă
Module	Un modul
Parameter	Un parametru
Property	O proprietate
ReturnValue	O valoare de retur
Struct	O structură

Aplicarea atributelor se face prin specificarea numelui lor între paranteze drepte; mai multe atribute fie se specificăunul deasupra celuilalt, fie în interiorul aceloraşi paranteze, despărţite prin virgulă. Exemplu:

```
[Serializable]
[Webservice]
```

echivalent cu:

```
[Serializable, Webservice]
```

În cazul atributelor ce se specifică pentru un assembly, forma lor este:

```
[assembly:AssemblyDelaySign(false)]
```

în cazul acestor din urmă atribute, specificarea lor se face după toate declaraţiile `using`, dar înaintea oricărui cod. În general, specificarea unui atribut se face prin scrierea lui imediat înaintea elementului asupra căruia se aplică:

```
using System;
[Serializable]
class ClasaSerializabila
{
//definitia clasei
}
```

12.1.2 Atribute predefinite

Tabelul 12.2 prezintă câteva dintre ele:

Tabelul 12.2: Atribute predefinite.

Atribut	Descriere
System.SerializableAttribute [Serializable]	Permite unei clase să fie serializată pe disc sau într–o reţea
System.NonSerializedAttribute [NonSerialized]	Permite unor membri să nu fie salvaţi pe reţea sau pe disc
System.Web.Services.WebServiceAttribute [WebService]	Permite specificarea unui nume şi a unei descrieri pentru un serviciu Web
System.Web.Services.WebMethodAttribute [WebMethod]	Marchează o metodă ca fiind expusă ca parte a unui serviciu Web
System.AttributeUsageAttribute [AttributeUsage]	Defineşte parametrii de utilizare pentru atribute
System.ObsoleteAttribute [Obsolete]	Marchează o secvenţă ca fiind scoasă din uz
System.Reflection.AssemblyVersionAttribute [AssemblyVersion]	Specifică numărul de versiune al unui assembly
System.Attribute.CLSCompliant [CLSCompliant]	Indică dacă un element de program este compatibil cu CLS
System.Runtime.InteropServices. DllImportAttribute [DllImport]	specifică locaţia DLL care conţine implementarea pentru o metodă externă

Între parantezele drepte se arată cum se specifică acel atribut.

Exemplul de mai jos foloseşte atributul *System.ObsoleteAttribute*.

```
using System;

namespace AttributeSample1
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            int s1 = AddTwoNumbers(2, 3);
            int s2 = AddNumbers(2, 3);
            int s3 = AddNumbers(2, 3, 4);
```

```
        }

        [Obsolete("obsolete: use AddNumbers instead")]
        static int AddTwoNumbers(int a, int b)
        {
            return a + b;
        }

        static int AddNumbers(params int[] numbers)
        {
            int result = 0;
            foreach(int number in numbers)
                result += number;
            return result;
        }
    }
}
```

La compilarea codului se generează un avertisment care semnalizează faptul că se utilizează o metodă care este scoasă din uz. Mesajul specificat ca parametru al atributului este afişat ca mesaj al avertismentului. Suplimentar, pentru atributul Obsolete se poate specifica dacă respectivul avertisment este sau nu interpretat ca o eroare, adăugând un parametru de tip bool: *false* ca la compilare să se genereze avertisment, *true* pentru ca utilizarea să fie tratată ca o eroare.

Exemplul de mai jos exemplifică serializarea şi deserializarea unui obiect:

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
[Serializable]
class Point2D
{
  public int X;
  public int Y;
}
class MyMainClass
{
  public static void Main()
  {
    Point2D My2DPoint = new Point2D();
    My2DPoint.X = 100;
```

```
    My2DPoint.Y = 200;
    Stream WriteStream = File.Create("Point2D.bin");
    BinaryFormatter BinaryWrite = new BinaryFormatter();
    BinaryWrite.Serialize(WriteStream, My2DPoint);
    WriteStream.Close();
    Point2D ANewPoint = new Point2D();
    Console.WriteLine("New Point Before Deserialization: ({0}, {1})",
    ANewPoint.X, ANewPoint.Y);
    Stream ReadStream = File.OpenRead("Point2D.bin");
    BinaryFormatter BinaryRead = new BinaryFormatter();
    ANewPoint = (Point2D)BinaryRead.Deserialize(ReadStream);
    ReadStream.Close();
    Console.WriteLine("New Point After Deserialization: ({0}, {1})",
    ANewPoint.X, ANewPoint.Y);
  }
}
```

12.1.3 Exemplificarea altor atribute predefinite

Atributul *Conditional*

Acest atribut se ataşează la o metodă pentru care se doreşte ca atunci când compilatorul o întâlneşte la un apel, dacă un anumit simbol nu e definit atunci nu va fi chemată. Este folosită pentru a omite anumite apeluri de metode (de exemplu cele folosite la etapa de debugging).

Exemplu:

```
using System;
using System.Diagnostics;

namespace CondAttrib
{
  class Thing
  {
    private string name;
    public Thing(string name)
    {
      this.name = name;
      SomeDebugFunc();
      SomeFunc();
    }
    public void SomeFunc()
```

```
    { Console.WriteLine("SomeFunc"); }

    [Conditional("DEBUG")]
    public void SomeDebugFunc()
    { Console.WriteLine("SomeDebugFunc"); }
  }

  public class Class1
  {
    [STAThread]
    static void Main(string[] args)
    {
      Thing t = new Thing("T1");
    }
  }
}
```

Definirea unui anumit simbol (de exemplu DEBUG) se poate face în două moduri:

- prin folosirea unei directive de preprocesor de tipul #define:

  ```
  #define DEBUG
  ```

- prin folosirea parametrilor din linia de comanda sau a posibilităţilor mediului integrat de dezvoltare. De exemplu, pentru a se defini un anumit simbolul din Visual Studio se procedează astfel: clic dreapta în Solution Explorer pe numele proiectului, selectare Properties apoi Configuration Properties. Pe linia Conditional Compilation Constants se adaugă simbolul dorit. Acest lucru va avea ca efect folosirea opţiunii *define* a compilatorului.

Dacă la compilare simbolul nu este definit atunci compilatorul va ignora apelurile de metode calificate cu atributul Conditional. Acest lucru se poate verifica atât urmărind execuţia programului, cât şi din inspecţia codului IL rezultat la compilare.

Atributul CLSCompliant

Acest atribut se aplică pe assembly–uri. Dacă un assembly este marcat ca fiind *CLSCompliant*, orice tip expus public în assembly care nu este compatibil CLS trebuie să fie marcat cu *CLSCompliant(false)*. De exmplu,

CLS prevede faptul că tipurile de date întregi ar trebui să fie cu semn. O clasă poate să conţină membri (câmpuri, metode) de tip unsigned, dar dacă respectiva clasă este declarată ca fiind CLSCompliant atunci acestea ar trebui declarate ca fiind ne-vizibile din afara assembly–ului. Dacă ele sunt expuse în afara assembly–ului, unele limbaje .NET s-ar putea să nu le poată folosi efetiv. Pentru a ajuta dezvoltatorii .NET să evite asemenea situaţii, platforma pune la dispoziţie atributul CLSCompliant cu care se controlează răspunsul compilatorului la expunerea entităţilor ne-compatibile cu CLS; astfel, se va genera o eroare sau se vor ignora asemenea cazuri.

Exemplu:

```
[assembly:CLSCompliant(true)]
namespace DemoCLS
{
    public class ComplianceTest
    {
    //Tipul uint nu ste compatibil CLS
    //deaorece acest camp este privat, regulile CLS nu se aplica
    private uint a = 4;

    //deoarece acest camp uint este public, avem
    //incompatibilitate CLS
    public uint B = 5;

    //Acesta este modul corect de expunere a unui uint:
    //tipul long este compatibil CLS
    public long A
    {
       get { return a; }
    }
    }
}
```

Dacă se compilează assembly-ul de mai sus atunci apare o eroare:

```
Type of 'DemoCLS.ComplianceTest.B' is not CLS-compliant
```

Pentru ca această eroare să nu mai apară putem să declarăm B ca fiind invizibil din exteriorul assembly-ului sau să adăugăm inaintea lui B atributul:

```
[CLSCompliant(false)]
```

Meritul acestui atribut este că anunţă programatorul despre eventualele neconcordanţe cu Common Language Specifications.

12.1.4 Atribute definite de utilizator

În general, atributele predefinite acoperă marea majoritate a situaţiilor care cer utilizarea de atribute. Eventualizatea ca utilizatorul să dorească crearea propriilor sale atribute este prevăzută de către platforma .NET, dându–se posibilitatea definirii lor.

Există câteva situaţii în care e benefică definirea de noi atribute. Exemplul cel mai des întâlnit este acela în care pentru a se menţine informaţii despre un cod la care lucrează mai multe echipe, se defineşte un atribut care să servească la pasarea de informaţie relativ la porţiuni din cod. Un alt exemplu este utilizarea unui sistem de urmărire a stadiului de dezvoltare a codului, care ar folosi informaţia stocată în atribute.

Un atribut utilizator este o clasă definită ca până acum. Se cere a fi derivată din *System.Attribute* fie direct, fie indirect. Sunt câţiva paşi care trebuie parcurşi pentru realizara unui atribut: specificarea ţintei, derivarea adecvată a clasei atribut, denumirea clasei în conformitate cu anumite reguli (recomandat, dar nu obligatoriu) şi definirea clasei.

Pasul 1. Declararea ţintei

Primul pas în crearea unui atribut utilizator este specificarea domeniului său de aplicabilitate, adică a elementelor cărora li se poate ataşa. Ţintele sunt cele din tabelul 12.1, care sunt de fapt valori din enumerarea *AttributeTargets*. Specificarea ţintei se face prin intermediul unui (meta)atribut *AttributeUsage*. Pe lângă ţinta propriu–zisă, se mai pot specifica valorile proprietăţilor *Inherited* şi *AllowMultiple*.

Proprietatea *Inherited* este de tip boolean şi precizează dacă atributul poate fi moştenit de către clasele derivate din cele căreia i se aplică. Valoarea implicită este *true*.

Proprietatea *AllowMultiple* este de asemenea de tip boolean şi specifică dacă se pot utiliza mai multe instanţe ale atributului pe un acelaşi element. Valoarea implicită este *false*.

Vom defini mai jos ţinta atributului pe care îl vom construi ca fiind assembly–ul, clasa, metoda, cu posibilitate de repetare a sa:

```
[AttributeUsage(AttributeTargets.Assembly|AttributeTargets.Class
      | AttributeTargets.Method, AllowMultiple=true)]
```

Pasul 2. Declararea unei clase atribut

Pentru declararea unei clase atribut, următoarele reguli trebuie să fie avute în vedere:

- (obligatoriu) O clasă atribut trebuie să deriveze direct sau indirect *System.Attribute*
- (obligatoriu) O clasă atribut trebuie să fie declarată ca fiind publică
- (recomandat) Un atribut ar trebui să aibă sufixul *Attribute*.

Vom declara clasa atribut *CodeTrackerAttribute*. Acest atribut s-ar putea folosi în cazul în care s-ar dori urmărirea dezvoltării codului în cadrul unui proiect de dimensiuni foarte mari, la care participă mai multe echipe. Acest atribut utilizator va fi inclus în codul compilat şi distribuit altor echipe (care nu vor avea acces la cod). O alternativă ar fi folosirea documentaţiei XML generate după comentariile din cod.

```
[AttributeUsage(AttributeTargets.Assembly|AttributeTargets.Class
       | AttributeTargets.Method, AllowMultiple=true)]
public class CodeTrackerAttribute : System.Attribute
{
    //cod
}
```

Pasul 3. Declararea constructorilor şi a proprietăţilor

Acest pas constă în definirea efectivă membrilor clasei. Vom considera cătributul pe care îl creăm va purta trei informaţii: numele programatorului care a acţionat asupra respectivei unităţi de cod, faza în care se află, opţional note. Primele două atribute sunt mandatorii şi vor fi preluate prin constructor, al treilea poate fi setat prin intermediul unei proprietăţi.

```
using System;
[AttributeUsage(AttributeTargets.Assembly|AttributeTargets.Class
       | AttributeTargets.Method, AllowMultiple=true)]
public class CodeTrackerAttribute : System.Attribute
{
    private string name;
    private string phase;
    private string notes;

    public CodeTrackerAttribute(string name, string phase)
    {
      this.name = name;
      this.phase = phase;
    }
```

```
    public virtual string Name
    {
      get{return name;}
    }

    public virtual string Phase
    {
      get{return phase;}
    }

    public virtual string Notes
    {
      get{return notes;}
      set{notes=value;}
    }
}
```

Pasul 4. Utilizarea atributului utilizator

Atributele definite de utilizator sunt folosite în acelaşi mod ca şi cele implicite. Codul de mai jso exemplifică acest lucru:

```
[CodeTracker("Lucian Sasu", "implementing specification",
  Notes = "First attempt")]
class AttribTest
{
  public AttribTest()
  {
    Console.WriteLine("AttribTest instance");
  }
  [CodeTracker("Lucian Sasu", "April 1st 2005")]
  public void SayHello(String message)
  {
    Console.WriteLine(message);
  }
}
```

O inspeţie a codului IL rezultat arată că aceste atribute s-au salvat în codul compilat. Obţinerea acestor atribute şi a valorilor lor se poate face foarte uşor prin reflectare.

12.2 Fire de execuţie

Firele de execuţie[1] sunt responsabile cu multitasking–ul în interiorul unei aplicaţii. Clasele şi interfeţele responsabile pentru crearea aplicaţiilor multifir se găsesc în spaţiul de nume *System.Threading*. Vom discuta în cele ce urmează despre managementul thread–urilor şi despre sincronizare.

De cele mai multe ori crearea de thread–uri este necesară pentru a da utilizatorului impresia că programul execută mai multe acţiuni simultan, în cadrul unei aplicaţii. Pentru ca o interfaţă utilizator să poată fi folosită fără a se aştepta încheierea unei anumite secvenţe de instrucţiuni, e nevoie de mai multe fire de execuţie (unele pentru procesarea efectivă a informaţiei, altele care să răspundă acţiunilor utilizatorului). Pentru probleme de calcul numeric, exprimarea procesului de calcul se poate face foarte natural sub formă de fire de execţie. De asemenea, strategia de rezolvare "divide et impera" se pretează natural la o lucrul cu fire de execuţie. În sfârşit, se poate beneficia de sisteme cu procesoare hyperthreading, multiprocesor, multicore sau combinaţii ale acestora.

12.3 Managementul thread–urilor

12.3.1 Pornirea thread–urilor

Cea mai simplă metodă de a crea un fir de execţie este de a crea o instanţă a clasei *Thread*, al cărei constructor preia un singur argument de tip delegat. În BCL este definit tipul delegat *ThreadStart*, care este folosit ca prototip pentru orice metodă care se vrea a fi lansată într–un fir de execuţie. Declaraţia de *ThreadStart* este:

```
public delegate void ThreadStart();
```

Crearea unui fir de execuţie se face pe baza unei metode care returnează *void* şi nu preia nici un parametru:

```
ThreadStart ts = new ThreadStart(MyFunc);
Thread myThread = new Thread( ts );
```

Un exemplu simplu este:

```
using System;
using System.Threading;
```

[1] Engl: threads; vom folosi alternativ termenii "thread" şi "fir de execuţie"

```
class SimpleThreadApp
{
    public static void WorkerThreadMethod()
    {
        Console.WriteLine("[WorkerThreadMethod] Worker " +
            "thread started");
    }

    public static void Main()
    {
        ThreadStart worker = new ThreadStart(WorkerThreadMethod);
        Console.WriteLine("[Main] Creating worker thread");
        Thread t = new Thread(worker);
        t.Start();
        Console.WriteLine( "[Main] Have requested the start of
            worker thread");
        Console.ReadLine();
    }
}
```

Până la linia `t.Start()` există un singur thread: cel dat de pornirea metodei `Main`. După `t.Start()` sunt 2 thread–uri: cel anterior şi `t`. Primul esaj din Main va fi tipărit înaintea celui din thread–ul t; în funcţie de cum anume se planifică thread–urile pentru execuţie, mesajul de după t.Start() poate să apară înainte sau după mesajul tipărit de metoda `WorkerThreadMethod()`. De remarcat că simpla creere a firului de execuţie nu determină şi pornirea lui: acest lucru se întâmplă dupa apelul metodei `Start` definită în clasa `Thread`.

Exemplul următor porneşte două fire de execuţie. Funcţiile care conţin codul ce se va executa în câte un thread sunt `Increment()` şi `Decrement()`:

```
using System;
using System.Threading;
class Tester
{
  static void Main( )
  {
    Tester t = new Tester( );
    t.DoTest( );
  }
  public void DoTest( )
```

```
  {
    // creeaza un thread pentru Incrementer
    Thread t1 = new Thread( new ThreadStart(Incrementer) );
    // creeaza un thread pentru Decrementer
    Thread t2 = new Thread( new ThreadStart(Decrementer) );
    // porneste threadurile
    t1.Start( );
    t2.Start( );
  }

  public void Incrementer( )
  {
    for (int i = 1;i<=1000;i++)
    {
      Console.WriteLine( "Incrementer: {0}", i);
    }
  }

  public void Decrementer( )
  {
    for (int i = 1000;i>0;i--)
    {
      Console.WriteLine("Decrementer: {0}", i);
    }
  }
}
```

La ieşire se vor mixa mesajele tipărite de primul thread cu cele tipărite de cel de–al doilea thread:

```
...
Incrementer: 102
Incrementer: 103
Incrementer: 104
Incrementer: 105
Incrementer: 106
Decrementer: 1000
Decrementer: 999
Decrementer: 998
Decrementer: 997
...
```

Perioada de timp alocată fiecărui thread este determinată de către planificatorul de fire de execuţie[2] şi depinde de factori precum viteza procesorului, gradul lui de ocupare, etc.

O altă modalitate de obţinere a unei referinţe la un thread este prin apelul proprietăţii statice `Thread.CurrentThread` pentru firul de execuţie curent. În exemplul de mai jos se obţine obiectul Thread asociat firului de execuţie curent, i se setează numele (proprietate read/write de tip String) şi se afişează pe ecran:

```
Thread current = Thread.CurrentThread;
current.Name = "My name";
Console.WriteLine("nume={0}", current.Name );
```

12.3.2 Metoda *Join()*

Există situaţii în care, înaintea unei instrucţiuni trebuie să se asigure faptul că un alt fir de execuţie t s-a terminat; acest lucru se va face folosind apelul *t.Join()* înaintea instrucţiunii în cauză; în acest moment firul de execuţie care a apelat `t.Join()` intră în aşteptare.

Dacă de exemplu în metoda *Main* se lansează o colecţie de thread–uri (stocată în *myThreads*), atunci pentru a se continua execuţia numai după ce toate firele din colecţie s–au terminat, se procedează astfel:

```
foreach( Thread myThread in myThreads )
{
  myThread.Join();
}
Console.WriteLine(‘‘All my threads are done’’);
```

Mesajul final se va tipări doar când toate firele de execuţie s–au terminat.

12.3.3 Suspendarea firelor de execuţie

Se poate ca în anumite cazuri să se dorească suspendarea unui fir de execuţie pentru o scurtă perioadă de timp. Clasa *Thread* oferă o metodă statică supraîncărcată *Sleep()*, care poate prelua un parametru de tip *int* reprezentând milisecundele de "adormire", iar a doua variantă preia un argument de tip *TimeSpan*, care reprezintă cea mai mică unitate de timp care poate fi specificată, egală cu 100 nanosecunde. Pentru a cere firului de execuţie curent să se suspende pentru o secundă, se execută în cadrul acestuia:

[2]Engl: thread scheduler

```
Thread.Sleep(1000);
```

În acest fel se semnalează planificatorului de fire de execuţie că poate lansa un alt thread.

Dacă în exemplul de mai sus se adaugă un apel *Thread.Sleep(1)* după fiecare *WriteLine()*, atunci ieşirea se schimbă dramatic:

```
Iesire (extras)
Incrementer: 0
Incrementer: 1
Decrementer: 1000
Incrementer: 2
Decrementer: 999
Incrementer: 3
Decrementer: 998
Incrementer: 4
Decrementer: 997
Incrementer: 5
Decrementer: 996
Incrementer: 6
Decrementer: 995
```

12.3.4 Omorârea thread–urilor

De obicei, un fir de execuţie moare după ce se termină de executat. Se poate totuşi cere unui fir de execuţie să îşi înceteze execuţia folosind metoda *Abort()*. Acest lucru va duce la aruncarea unei excepţii în interiorul firului de execuţie căruia i se cere suspendarea: *ThreadAbortedException*, pe care firul respectiv o poate prinde şi procesa, permiţându–i eliberarea de resurse alocate. Ca atare, se recomandă ca o metodă care se va lansa ca fir de execuţie să se compună dintr–un bloc *try* care conţine instrucţiunile utile, după care un bloc *catch* şi/sau *finally* care vor efectua eliberarea de resurse.

Exemplu:

```
using System;
using System.Threading;
class Tester
{
  static void Main( )
  {
    Tester t = new Tester( );
    t.DoTest( );
```

```
  }
  public void DoTest( )
  {
    // creeaza un vector de threaduri
    Thread[] myThreads =
    {
      new Thread( new ThreadStart(Decrementer) ),
      new Thread( new ThreadStart(Incrementer) ),
      new Thread( new ThreadStart(Incrementer) )
    };
    // porneste fiecare thread
    int ctr = 1;
    foreach (Thread myThread in myThreads)
    {
      myThread.IsBackground=true;
      myThread.Start( );
      myThread.Name = "Thread" + ctr.ToString( );
      ctr++;
      Console.WriteLine("Started thread {0}", myThread.Name);
      Thread.Sleep(50);
    }
    // dupa ce firele se pornesc,
    // comanda oprirea threadului 1
    myThreads[1].Abort( );
    // asteapta ca fiecare thread sa se termine
    foreach (Thread myThread in myThreads)
    {
      myThread.Join( );
    }
    Console.WriteLine("All my threads are done.");
  }
  // numara descrescator de la 1000
  public void Decrementer( )
  {
    try
    {
      for (int i = 1000;i>0;i--)
      {
        Console.WriteLine(‘‘Thread {0}. Decrementer: {1}’’,
        Thread.CurrentThread.Name, i);
        Thread.Sleep(1);
```

```
      }
    }
    catch (ThreadAbortedException)
    {
      Console.WriteLine(
      ‘‘Thread {0} interrupted! Cleaning up...’’,
      Thread.CurrentThread.Name);
    }
    finally
    {
      Console.WriteLine(‘‘Thread {0} Exiting. ’’,
      Thread.CurrentThread.Name);
    }
  }
  // numara cresacator pana la 1000
  public void Incrementer( )
  {
    try
    {
      for (int i =1;i<=1000;i++)
      {
        Console.WriteLine(‘‘Thread {0}. Incrementer: {1}’’,
        Thread.CurrentThread.Name, i);
        Thread.Sleep(1);
      }
    }
    catch (ThreadAbortedException)
    {
      Console.WriteLine( ‘‘Thread {0} interrupted! Cleaning up...’’,
      Thread.CurrentThread.Name);
    }
    finally
    {
      Console.WriteLine( ‘‘Thread {0} Exiting. ’’,
      Thread.CurrentThread.Name);
    }
  }
}
```

Ieşire:

```
Started thread Thread1
```

```
Thread Thread1. Decrementer: 1000
Thread Thread1. Decrementer: 999
Thread Thread1. Decrementer: 998
Started thread Thread2
Thread Thread1. Decrementer: 997
Thread Thread2. Incrementer: 0
Thread Thread1. Decrementer: 996
Thread Thread2. Incrementer: 1
Thread Thread1. Decrementer: 995
Thread Thread2. Incrementer: 2
Thread Thread1. Decrementer: 994
Thread Thread2. Incrementer: 3
Started thread Thread3
Thread Thread1. Decrementer: 993
Thread Thread2. Incrementer: 4
Thread Thread2. Incrementer: 5
Thread Thread1. Decrementer: 992
Thread Thread2. Incrementer: 6
Thread Thread1. Decrementer: 991
Thread Thread3. Incrementer: 0
Thread Thread2. Incrementer: 7
Thread Thread1. Decrementer: 990
Thread Thread3. Incrementer: 1
Thread Thread2 interrupted! Cleaning up...
Thread Thread2 Exiting.
Thread Thread1. Decrementer: 989
Thread Thread3. Incrementer: 2
Thread Thread1. Decrementer: 988
Thread Thread3. Incrementer: 3
Thread Thread1. Decrementer: 987
Thread Thread3. Incrementer: 4
Thread Thread1. Decrementer: 986
Thread Thread3. Incrementer: 5
// ...
Thread Thread1. Decrementer: 1
Thread Thread3. Incrementer: 997
```

12.3.5 Sugerarea priorităţilor firelor de execuţie

Un fir de execuţie se lansează implicit cu prioritatea *ThreadPriorityLevel.Normal*. Dar scheduler–ul poate fi influenţat în activitatea sa prin setarea de diferite

nivele de prioritate pentru fire; aceste nivele fac parte din enumerarea *ThreadPriorityLevel*: *ThreadPriorityLevel.TimeCritical, ThreadPriorityLevel.Highest, ThreadPriorityLevel.AboveNormal, ThreadPriorityLevel.Normal, ThreadPriorityLevel.BelowNormal, ThreadPriorityLevel.Lowest, ThreadPriorityLevel.Idle*. Prioritatea este descrescătoare în lista prezentată. Pe baza priorităţii procesului care conţine firele de execuţie şi a priorităţii firelor, se calculează un nivel de prioritate (de ex. pe maşini Intel valori între 0 şi 31) care determină prioritatea în ansamblul sistemului de operare a firului respectiv.

Setarea unei anumite priorităţi se face folosind proprietatea *Priority*:

```
myThread.Priority = ThreadPriorityLevel.Highest;
```

12.3.6 Fire în fundal şi fire în prim-plan

Relativ la proprietatea booleană `IsBackground`, trebuie făcută precizarea că un fir de execuţie poate să se execute în fundal (background) sau în prim plan (foreground). Diferenţa dintre cele două posibilităţi o constituie faptul că dacă un proces are măcar un fir de execuţie în foreground, CLR va menţine aplicaţia în execuţie. O dată ce toate firele de execuţie de tip foreground se termină, CLR va executa Abort() pentru fiecare fir de execuţie de tip background (dacă mai există aşa ceva) şi termină procesul.

Exemplu:

```
using System;
using System.Threading;

class Test
{
    static void Main()
    {
        BackgroundTest shortTest = new BackgroundTest(10);
        Thread foregroundThread =
            new Thread(new ThreadStart(shortTest.RunLoop));
        foregroundThread.Name = "ForegroundThread";

        BackgroundTest longTest = new BackgroundTest(50);
        Thread backgroundThread =
            new Thread(new ThreadStart(longTest.RunLoop));
        backgroundThread.Name = "BackgroundThread";
        backgroundThread.IsBackground = true;

        foregroundThread.Start();
```

```
        backgroundThread.Start();
    }
}

class BackgroundTest
{
    int maxIterations;

    public BackgroundTest(int maxIterations)
    {
        this.maxIterations = maxIterations;
    }

    public void RunLoop()
    {
        String threadName = Thread.CurrentThread.Name;

        for(int i = 0; i < maxIterations; i++)
        {
            Console.WriteLine("{0} count: {1}",
                threadName, i.ToString());
            Thread.Sleep(250);
        }
        Console.WriteLine("{0} finished counting.", threadName);
    }
}
```

Firul din foreground va menţine procesul în execuţie până când se termină ciclul său `while`. Când acesta se termină, procesul este oprit, chiar dacă ciclul `while` din firul de execuţie din background nu şi-a terminat execuţia.

12.4 Sincronizarea

Sincronizarea se ocupă cu controlarea accesului la resurse partajate de mai multe fire de execuţie. De exemplu, se poate cere ca utilizarea unei resurse anume să se facă la un moment dat de către un singur fir de execuţie. Vom discuta aici trei mecanisme de sincronizare: clasa *Interlock*, instrucţiunea C# *lock* şi clasa *Monitor*. Exemplele se vor baza pe acces la o resursă partajată, ca mai jos:

```
public void Incrementer( )
```

```
{
  try
  {
    while (counter < 1000)
    {
      int temp = counter;
      temp++; // increment
      // simuleza o sarcina oarecare in acest thread
      Thread.Sleep(1);
      // atribuie valoarea incrementata
      // variabilei counter
      // si afiseaza rezultatul
      counter = temp;
      Console.WriteLine( ‘‘Thread {0}. Incrementer: {1}’’,
      Thread.CurrentThread.Name, counter);
    }
  }
  catch (ThreadAbortedException)
  {
    Console.WriteLine( ‘‘Thread {0} interrupted! Cleaning up...’’,
    Thread.CurrentThread.Name);
  }
  finally
  {
    Console.WriteLine( ‘‘Thread {0} Exiting. ’’,
    Thread.CurrentThread.Name);
  }
}
```

Câmpul *counter* se iniţializează cu 0. Să presupunem că pornim două fire de execuţie pe baza metodei *Incrementer()* de mai sus. Este posibil să se întâmple următoarele: primul thread va citi valoarea lui *counter* (0) şi o va atribui unei variabile temporare, pe care o va incrementa apoi. Al doilea fir se activează şi el, va citi valoarea (nemodificată) lui *counter* şi va atribui această valoare unei variabile temporare. Primul fir de execuţie îşi termină munca, apoi asignează valoarea variabilei temporare (1) lui *counter* şi o afişează. Al doilea thread face exact acelaşi lucru. Se tipăreşte astfel 1, 1. La următoarele iteraţii se va afişa 2, 2, 3, 3, etc, în locul lui 1, 2, 3, 4.

Exemplu:

```
using System;
using System.Threading;
```

```
class Tester
{
  private int counter = 0;
  static void Main( )
  {
    Tester t = new Tester( );
    t.DoTest( );
  }
  public void DoTest( )
  {
    Thread t1 = new Thread( new ThreadStart(this.Incrementer) );
    t1.IsBackground=true;
    t1.Name = ‘‘ThreadOne’’;
    t1.Start( );
    Console.WriteLine(‘‘Started thread {0}’’, t1.Name);
    Thread t2 = new Thread( new ThreadStart(this.Incrementer) );
    t2.IsBackground=true;
    t2.Name = ‘‘ThreadTwo’’;
    t2.Start( );
    Console.WriteLine(‘‘Started thread {0}’’, t2.Name);
    t1.Join( );
    t2.Join( );
  }
  // numara crescator pana la 1000
  public void Incrementer( )
  {
    //la fel ca la inceputul sectiunii
  }
}
```

Ieşire:

```
Started thread ThreadOne
Started thread ThreadTwo
Thread ThreadOne. Incrementer: 1
Thread ThreadOne. Incrementer: 2
Thread ThreadOne. Incrementer: 3
Thread ThreadTwo. Incrementer: 3
Thread ThreadTwo. Incrementer: 4
Thread ThreadOne. Incrementer: 4
Thread ThreadTwo. Incrementer: 5
Thread ThreadOne. Incrementer: 5
```

```
Thread ThreadTwo. Incrementer: 6
Thread ThreadOne. Incrementer: 6
```

Trebuie deci să se realizeze o excludere reciprocă a thread–urilor pentru accesul la *counter*.

12.4.1 Clasa *Interlocked*

Incrementarea şi decrementarea unei valori este o situaţie atât de des întâlnită, încât C# pune la dispoziţie o clasă specială *Interlocked* pentru o rezolvare rapidă. Clasa include două metode statice, *Increment()* şi *Decrement()*, care incrementează sau decrementează o valoare, însă sub un control sincronizat.

Putem modifica metoda *Incrementer()* de mai sus astfel:

```
public void Incrementer( )
{
  try
  {
    while (counter < 1000)
    {
      Interlocked.Increment(ref counter);
      // simuleaza o sarcina in aceasta metoda
      Thread.Sleep(1);
      // asigura valoarea decrementata
      // si afiseaza rezultatul
      Console.WriteLine(
      "Thread {0}. Incrementer: {1}",
      Thread.CurrentThread.Name,
      counter);
    }
  }
  //blocurile catch si finally raman neschimbate
}
```

Ieşirea este cea dorită:

```
Started thread ThreadOne
Started thread ThreadTwo
Thread ThreadOne. Incrementer: 1
Thread ThreadTwo. Incrementer: 2
Thread ThreadOne. Incrementer: 3
Thread ThreadTwo. Incrementer: 4
```

```
Thread ThreadOne. Incrementer: 5
Thread ThreadTwo. Incrementer: 6
Thread ThreadOne. Incrementer: 7
Thread ThreadTwo. Incrementer: 8
```

12.4.2 Instrucţiunea *lock*

Există situaţii când vrem să blocăm alte variabile decât cele de tip int. Un lock marchează o secţiune critică a codului, producând astfel sincronizare pentru un obiect. La utilizare, se specifică un obiect pentru care se stabileşte un lock, după care o instruţiune sau un grup de instrucţiuni. Lock–ul este înlăturat la sfârşitul instrucţiunii/blocului de instruţiuni. Sintaxa este:

```
lock(expresie)
{
  instructiuni
}
```

Exemplu: metoda *Incrementer()* se va modifica după cum urmează:

```
public void Incrementer( )
{
  try
  {
    while (counter < 1000)
    {
      lock (this)
      {
        int temp = counter;
        temp++;
        Thread.Sleep(1);
        counter = temp;
      }
      // atribuie valoarea decrementata
      // si afiseaza rezultatul
      Console.WriteLine( "Thread {0}. Incrementer: {1}",
      Thread.CurrentThread.Name, counter);
    }
  }
  //blocurile catch si finally raman neschimbate
}
```

Rezultatele sunt afişate exact ca la secţiunea 12.4.1.

12.4.3 Clasa *Monitor*

Clasa *Monitor* conţine metode pentru a controla sincronizarea firelor de execuţie, permiţând declararea unei zone critice în care la un moment dat doar un thread trebuie să opereze.

Atunci când se doreşte să se înceapă sincronizarea, se va apela metoda *Enter*, dând obiectul pentru care se va face blocarea:

```
Monitor.Enter( obiect );
```

Dacă monitorul este nedisponibil, atunci înseamnă că un alt thread este într–o regiune critică a obiectului respectiv. Se mai poate folosi de asemenea metoda *Wait()*, care eliberează monitorul, dar blochează threadul, informând CLR că atunci când monitorul devine din nou liber, thread–ul curent ar vrea să îşi continue execuţia (este adăugat într–o coadă de aşteptare formată din fire de execuţie blocate pe obiect). Terminarea zonei critice se face folosind metoda *Exit()* a clasei *Monitor*. Metoda *Pulse()* semnalează că a avut loc o schimbare de stare, în urma căreia este posibil ca un alt fir de execuţie care aşteaptă va putea să fie continuat (ordinea de selectare a firelor ce vor fi executate fiind ordinea introducerii în coada de aşteptare). Înrudită este metoda *PulseAll* care anunţă toate obiectele blocate pe un anumit obiect de schimbarea de stare.

Să presupunem că avem o clasă *MessageBoard* unde fire individuale pot citi şi scrie mesaje. Vom sincroniza accesul la această clasă astfel încât doar un thread să poată acţiona la un moment dat. Clasa *MessageBoard* va avea o metodă *Reader()* şi una *Writer()*.

Metoda *Reader()* determină dacă string–ul *message* conţine vreun mesaj valabil, iar metoda *Writer()* va scrie în acest string. Dacă nu sunt mesaje în timpul citirii, thread–ul *Reader()* va intra în stare de aşteptare folosind *Wait()* până când metoda *Writer()* scrie un mesaj şi transmite un mesaj via *Pulse()* pentru a trezi alte thread–uri.

```
using System;
using System.Threading;
class MessageBoard
{
  private String messages = ‘‘no messages’’ ;
  public void Reader()
  {
    try
    {
      Monitor.Enter(this);
```

```
      //daca nu e nici un mesaj atunci asteapta
      if (messages == ‘‘no messages’’)
      {
        Console.WriteLine(‘‘{0} {1}’’,
        Thread.CurrentThread.Name, messages);
        Console.WriteLine(‘‘{0} waiting...’’,
        Thread.CurrentThread.Name);
        Monitor.Wait(this);
      }
      //inseamna ca mesajul s-a schimbat
      Console.WriteLine(‘‘{0} {1}’’,
      Thread.CurrentThread.Name, messages);
    }
    finally
    {
      Monitor.Exit(this);
    }
  }
  public void Writer()
  {
    try
    {
      Monitor.Enter(this);
      messages = ‘‘Greetings Caroline and Marianne!’’;
      Console.WriteLine(‘‘{0} Done writing message...’’,
      Thread.CurrentThread.Name);
      //semnaleaza threadului de asteptare ca s-a schimbat mesajul
      Monitor.Pulse(this);
    }
    finally
    {
      Monitor.Exit(this);
    }
  }
  public static void Main()
  {
    MessageBoard myMessageBoard = new MessageBoard();
    Thread reader = new Thread(new
    ThreadStart(myMessageBoard.Reader));
    reader.Name = ‘‘ReaderThread:’’;
    Thread writer = new Thread( new
```

```
    ThreadStart(myMessageBoard.Writer));
    writer.Name = ‘‘WriterThread:’’;
    reader.Start();
    writer.Start();
  }
}
```

Ieşirea este:

```
ReadrThread: no messages
ReaderThread: waiting...
WriterThread: Done writing message...
ReaderThread: Greetings Caroline and Marianne!
```

După cum se vede mai sus, thread–ul *reader* este pornit primul, el blochează clasa obeictul de tip *MessageBoard*, ceea ce înseamnă că are acces exclusiv la variabila *message*. Dar deoarece *message* nu conţine nici un mesaj, thread-ul *reader* eliberează obiectul pe care l–a blocat mai înainte prin apelul *Wait()*. Thread–ul *writer* va putea acum să blocheze obiectul pentru a scrie mesajul. Apoi el cheamă metoda *Pulse()* pentru a semnala firului *reader* că a apărut o schimbare de stare a obiectului indicat wde *this*.

Curs 13

Noutăţi în C# 4.0

13.1 Parallel Linq

Parallel Linq (PLINQ) permite executarea de interogări în mod paralel, pentru stituaţiile în care există un sistem de tip multioprocesor, multicore sau cu suport de hyperthreading. Pentru a transforma o interogare clasică într–una paralelizată, trebuie adăugată specificarea `AsParallel` la sursa de date peste care se execută interogarea. Mai exact, plecând de la interogarea Linq:

```
var result =
  from x in source
  where [conditie]
  select [ceva]
```

se obţine interogarea PLINQ:

```
var result =
  from x in source.AsParallel()
  where [conditie]
  select [ceva]
```

Echivalent, se poate face transformarea folosind metode Linq:

```
//varianta neparalela
var result =  source.Where(x => [conditie]).Select(x => [ceva]);
//varianta cu paralelism
var result =  source.AsParallel().Where(x => [conditie]).Select(x => [ceva]);
```

Prin adăugarea acestei metode `AsParallel()` se folosesc metode din clasa `ParallelEnumerable`, în timp ce în Linq-ul clasic se folosesc metode din clasa `Enumerable`.

Prezentăm următorul exemplu preluat din [9]:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;

namespace TestPLINQ
{
    class Program
    {
        static void Main()
        {
            Stopwatch sw = new Stopwatch();
            sw.Start();
            DoIt();
            sw.Stop();
            Console.WriteLine("Elapsed = " +
              sw.ElapsedMilliseconds.ToString());
        }

        private static bool isPrime(int p)
        {
            int upperBound = (int)Math.Sqrt(p);
            for (int i = 2; i <= upperBound; i++)
            {
                if (p % i == 0) return false;
            }
            return true;
        }

        static void DoIt()
        {
            IEnumerable<int> arr = Enumerable.Range(2, 10000000);
            var q =
              from n in arr
              where isPrime(n)
              select n.ToString();
            IList list = q.ToList();
```

```
            Console.WriteLine(list.Count.ToString());
        }
    }
}
```

Pentru un sistem oarecare, timpul mediu de rulare este de aproximativ 17 secunde; rescriind interogarea pentru obţinerea variabilei `q` astfel:

```
var q =
      from n in arr.AsParallel()
      where isPrime(n)
      select n.ToString();
```

timpul mediu obţinut este de aproximativ 10 secunde.

Se pot configura aspecte precum gradul de paralelism, controlul ordinii, opţiuni de utilizare de buffere, dacă anumite porţiuni să se ruleze secvenţial etc prin folosirea metodelor de extensie precum `AsOrdered`, `AsUnordered` sau prin folosirea enumerării `ParallelQueryMergeOptions`.

13.2 Parametri cu nume şi parametri opţionali

Parametrii opţionali permit precizarea unor valori implicite pentru parametrii unor metode; dacă aceştia nu sunt precizaţi la apel, atunci valorile trimise metodei sunt cele declarate implicit.

Exemplu:

```
class Program
{
    static void Main(string[] args)
    {
        MyMethod(3);
        MyMethod(3, 4);
    }

    private static void MyMethod(int p, int q = 100)
    {
        Console.WriteLine("p= {0}, q={1}", p, q);
    }
}
```

Deşi avem o singură metodă, aceasta suportă cele două apeluri. Se poate chiar să avem mai mulţi de un parametru cu valoare implicită:

```
private static void MyMethod(int p=1, int q = 100)
{
  Console.WriteLine("p= {0}, q={1}", p, q);
}
```

În cazul în care avem măcar un astfel de parametru cu valoarea implicită, acesta trebuie să fie prezent după parametrii cu valori obligatorii. Astfel, încercarea de a scrie:

```
private void MyMethod(int p=1, int q){...}
```

duce la apariţia erorii de compilare:

```
Optional parameters must appear after all required parameters
```

Valorile furnizate pentru parametrii cu valori implicite trebuie să fie constante sau să aibe valori de forma `default(T)`.

În ceea ce priveşte folosirea parametrilor cu nume, să presupunem că avem următoarea metodă:

```
public void M(int x, int y = 5, int z = 7){...}
```

Dacă vrem ca la un apel să nu precizăm valoarea lui `y`, dar să o precizăm pe a lui `z`, am fi tentaţi să folosim:

```
M(1, ,-1)
```

ceea ce în cazul în care ar fi permis, ar duce la un cod greu de citit, în care abilitatea de numărare a virgulelor ar fi crucială. În locul acestei variante, s-a introdus posibilitatea de a preciza parametrii prin nume:

```
M(1, z:3);
//sau
M(x:1, z:3)
//sau chiar:
M(z:3, x:1)
```

Ordinea de evaluare a expresiilor date pentru parametri este dată de ordinea de precizare a numelor parametrilor, deci în ultimul exemplu expresia 3 este evaluată înainte de expresia 1. Mai precizăm şi că parametrii opţionali şi cu nume se pot folosi şi pentru constructori sau indexatori.

13.3 Tipuri de date dinamice

Tipurile dinamice permit tratarea unui obiect fără a fi crispaţi de provenienţa acestuia: obiect creat clasic, sau prin COM sau prin reflectare. Unui astfel de obiect i s pot transmite mesaje (=apeluri de metode), iar legitimitatea acestor apeluri este verificată la rulare. Altfel zis, pentru tipurile de date dinamice se renunţă la tipizarea statică specifică platformelor .NET de versiune anterioară, dar cu riscul de a primi erori doar la rularea aplicaţiei.

Acest tip de date dinamic se declară prin cuvântul cheie `dynamic`. Plecând de la un astfel de obiect, se pot apela diferite metode:

```
dynamic x = MyMethod();
x.AnotherMethod(3);
```

La rulare se verifică dacă tipul de date aferent variabilei `x` are metoda `AnotherMethod` care să permită apel cu un parametru de tip întreg.

Exemplu:

```
static void Main(string[] args)
{
    dynamic x = "abc";
    x = 3;
    Console.WriteLine(x.CompareTo(10));//metoda CompareTo este din
    //tipul System.Int32
}
```

Remarcăm că tipul unei variabile dinamice nu este setat odată pentru totdeauna. Alt exemplu este:

```
class ExampleClass
{
    public ExampleClass() { }
    public ExampleClass(int v) { }

    public void exampleMethod1(int i) { }

    public void exampleMethod2(string str) { }
}

//....
static void Main(string[] args)
{
    ExampleClass ec = new ExampleClass();
```

```
    // The following line causes a compiler error if exampleMethod1 has only
    // one parameter.
    //ec.exampleMethod1(10, 4);

    dynamic dynamic_ec = new ExampleClass();
    // The following line is not identified as an error by the
    // compiler, but it causes a run-time exception.
    dynamic_ec.exampleMethod1(10, 4);

    // The following calls also do not cause compiler errors, whether
    // appropriate methods exist or not.
    dynamic_ec.someMethod("some argument", 7, null);
    dynamic_ec.nonexistentMethod();
}
```

Se pot declara ca fiind dinamici si parametrii unei metode:

```
public static void Log(dynamic x)
{
  Console.WriteLine(x.ToString());
}

static void Main(string[] args)
{

  var x = 3;
  string y = "abc";

  Log(x);
  Log(y);
}
```

13.4 COM Interop

COM Interop este o facilitate existentă în versiuni mai vechi ale lui .NET framework care permite codului managed să apeleze cod unmanaged de tip Component Object Model, cum ar fi aplicaţiile Word sau Excel. De exemplu, pentru utilizarea unei celule de Excel, varianta veche de cod ce trebuia scrisă era:

```
((Excel.Range)excel.Cells[1, 1]).Value2 = "Hello";
```

pe când în .NET 4 a se scrie mai inteligibil astfel:

```
excel.Cells[1, 1].Value = "Hello";
```

sau în loc de

```
Excel.Range range = (Excel.Range)excel.Cells[1, 1];
```

se scrie:

```
Excel.Range range = excel.Cells[1, 1];
```

Exemplu:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Excel = Microsoft.Office.Interop.Excel;

namespace TestOffice
{
    class Program
    {
        static void Main(string[] args)
        {
            var excel = new Excel.Application();
            // Make the object visible.
            excel.Visible = true;

            // Create a new, empty workbook and add it to the collection returned
            // by property Workbooks. The new workbook becomes the active workbook.
            // Add has an optional parameter for specifying a praticular template.
            // Because no argument is sent in this example, Add creates a new workbo
            excel.Workbooks.Add();

            excel.Cells[1, 1].Value = "Hello";

            var processes = Process.GetProcesses()
            .OrderByDescending(p => p.WorkingSet64)
            .Take(10);
            int i = 2;
            foreach (var p in processes)
```

```
            {
                excel.Cells[i, 1].Value = p.ProcessName; // no casts
                excel.Cells[i, 2].Value = p.WorkingSet64;  // no casts
                i++;
            }

            Excel.Range range = excel.Cells[1, 1];       // no casts
            dynamic chart = excel.ActiveWorkbook.Charts.
                Add(After: excel.ActiveSheet);         // named and optional
            chart.ChartWizard(
                Source: range.CurrentRegion,
                Title: "Memory Usage in " + Environment.MachineName); //named
            chart.ChartStyle = 45;
            chart.CopyPicture(Excel.XlPictureAppearance.xlScreen,
                Excel.XlCopyPictureFormat.xlBitmap,
                Excel.XlPictureAppearance.xlScreen);

        }
    }
}
```

13.5 Covarianţă şi contravarianţă

Să presupunem că avem secvenţa de cod:

```
var strList = new List<string>();
List<object> objList = strList;
```

Linia a doua, dacă ar fi permisă, ar predispune la erori, deoarece s-ar permite mai departe:

```
objList.Add(new MyClass());
```

deci s–ar încălca punctul de plecare pentru colecţie: elementele ar trebui să fie toate de tip `String`. Ca atare, acest lucru nu ar avea sens să fie permis.

Începând cu .NET 4, se permite însă conversia către *interfeţe generice sau delegaţi generici* pentru care tipul generic este mai general decât argumentul generic dinspre care se face conversie. De exemplu, se poate scrie:

```
IEnumerable<object> myCollection = strList;
```

Aceasta este covarianţa şi se obţine prin declaraţiile:

```
public interface IEnumerable<out T> : IEnumerable
{
  IEnumerator<T> GetEnumerator();
}
public interface IEnumerator<out T> : IEnumerator
{
  bool MoveNext();
  T Current { get; }
}
```

unde cuvântul `out` are un alt sens decât la transmiterea de parametri: în C# 4.0,în acest context, semnifică faptul că tipul `T` poate să apară doar în poziţia de ieşire a unei metode din interfaţă. Interfaţa `IEnumerable` devine astfel covariantă în T şi se poate face conversie de la `IEnumerable<B>` la `IEnumerable<A>` pentru tipul `B` derivat din `A`.

Acest lucru este util pentru o situaţie de genul:

```
var result = strings.Union(objects);
```

unde se face reuniune între o coleţie de string–uri şi una de obiecte.

Contravarianţa permite conversii în sens invers ca la covarianţă. De exemplu, prin declaraţia:

```
public interface IComparer<in T>
{
public int Compare(T left, T right);
}
```

prin arată contravarianţă a tipului `T`, deci se poate face ca un `IComparer<object>` să fie considerat drept un `IComparer<String>`. Are sens, deoarece dacă un IComparer poate să compare orice două obiecte, atunci cu siguranţă poate să compare şi doua string-uri.

Curs 14

Fluxuri

Pentru aplicaţiile reale, datele care se prelucrează nu se mai preiau din tastatură, ci se accesează din fişiere de date. C#, precum alte limbaje anterioare, pune la dispoziţie o abstractizare numită "flux"[1] care permite manipularea datelor, indiferent de sursa acestora. Într–un astfel de flux, pachete de date urmează unele după altele.

În C# fişierele şi directoarele se manipulează prin intermediul unor clase predefinite. Acestea permit crearea, redenumirea, manipularea, ştergerea fişierelor şi a directoarelor. Manipularea *conţinutului* fişierelor se face prin intermediul stream-urilor cu sau fără buffer; de asemenea există un suport puternic pentru stream–uri asincrone (prelucrarea unui fişier se face de către un fir de execuţie creat automat). Datorită abstractizării, nu există diferenţe mari între lucrul cu fişiere aflate pe discul local şi datele aflate pe reţea; ca atare se va vorbi despre fluxuri bazate pe protocoale TCP/IP sau web. În sfârşit, *serializarea* este legată de fluxuri, întrucât permit "îngeţarea" unui obiect (care poate fi urmată de transportul lui pe reţea).

14.1 Sistemul de fişiere

Clasele care se folosesc pentru manipularea fişierelor şi a directoarelor se află în spaţiul de nume *System.IO*. Funcţionalitatea lor este aceeaşi cu a comenzilor disponibile într–un sistem de operare: creare, ştegere, redenumire, mutare de fişiere sau directoare, listarea conţinutului unui director, listarea atributelor sau a diferiţilor timpi pentru fişiere sau directoare, etc.

Clasele pe care vom folosi sunt: *Directory, DirectoryInfo, File, FileInfo.*

[1] Engl: Stream

14.1.1 Lucrul cu directoarele: clasele *Directory* şi *DirectoryInfo*

Clasa *Directory* conţine metode statice pentru crearea, mutarea, explorarea directoarelor. Clasa *DirectoryInfo* conţine doar membri nestatici şi permite aflarea diferitelor informaţii pentru un director anume.

Tabelul 14.1 conţine principalele metode ale clasei *Directory*, iar tabelul 14.2 conţine metodele notabile din clasa *DirectoryInfo*.

Tabelul 14.1: Metode ale clasei *Directory*.

Metoda	Explicaţie
CreateDirectory()	Creează directoarele şi subdirectoarele specificate prin parametru
Delete()	Şterge un director şi conţinutul său
Exists()	Returnează *true* dacă stringul specificat reprezintă numele unui director existent
GetCreationTime() SetCreationTime()	Returnează / setează data şi timpul creării unui director
GetCurrentDirectory() SetCurrentDirectory()	returnează / setează directorul curent
GetDirectories()	Returnează un şir de nume de subdirectoare
GetDirectoryRoot()	Returnează numele rădăcinii unui director specificat
GetFiles()	Returnează un şir de string–uri care conţine numele fişierelor din directorul specificat
GetLastAccesTime() SetLastAccesTime()	returnează / setează timpul ultimului acces pentru un director
GetLastWriteTime() SetLastWriteTime()	Returnează / setează timpul când directorul specificat a fost ultima oară modificat
GetLogicalDrives()	Returnează numele tuturor unităţilor logice sub forma drive:\
GetParent()	Returnează directorul curent pentru calea specificată
Move()	Mută un director (cu conţinut) într–o cale specificată

Tabelul 14.2: Metode ale clasei *DirectoryInfo*.

Metoda sau proprietate	Explicaţie
Attributes	Returnează sau setează atributele fişierului curent
CreationTime	Returnează sau setează timpul de creare al fişierului curent

Tabelul 14.2(continuare)

Metoda sau proprietate	Explicaţie
Exists	*true* dacă directorul există
Extension	Extensia directorului
FullName	Returnează calea absolută a fişierului sau a directorului
LastAccessTime	Returnează sau setează timpul ultimului acces
LastWriteTime	Returnează sau setează timpul ultimei scrieri
Parent	Directorul părinte al directorului specificat
Root	Rădăcina căii corespunzătoare
Create()	Creează un director
CreateSubdirectory()	Creează un subdirector în calea specificatu a
Delete()	Şterge un *DirectoryInfo* şi conţinutul său
GetDirectories()	Returnează un vector de tip *DirectoryInfo* cu subdirectoare
GetFiles()	Returnează lista fişierelor din director
MoveTo()	Mută un *DirectoryInfo* şi conţinutul său într–un nou loc

Exemplul următor foloseşte clasa *DirectoryInfo* pentru a realiza explorarea recursivă a unui director cu enumerarea tuturor subdirectoarelor conţinute. Se creează un obiect de tipul pomenit pe baza numelui de director de la care se va începe explorarea. O metodă va afişa numele directorului la care s–a ajuns, după care se apelează recursiv metoda pentru fiecare subdirector (obţinut via *GetDirectories()*).

```
using System;
using System.IO;
class Tester
{
  public static void Main( )
  {
    Tester t = new Tester( );
    //choose the initial subdirectory
    string theDirectory = @''c:\WinNT'';
    // call the method to explore the directory,
    // displaying its access date and all
    // subdirectories
    DirectoryInfo dir = new DirectoryInfo(theDirectory);
    t.ExploreDirectory(dir);
```

```
    // completed. print the statistics
    Console.WriteLine(‘‘\n\n{0} directories found.\n’’, dirCounter);
  }
  // Set it running with a directoryInfo object
  // for each directory it finds, it will call
  // itself recursively
  private void ExploreDirectory(DirectoryInfo dir)
  {
    indentLevel++; // push a directory level
    // create indentation for subdirectories
    for (int i = 0; i < indentLevel; i++)
    Console.Write(" "); // two spaces per level
    // print the directory and the time last accessed
    Console.WriteLine("[{0}] {1} [{2}]\n",
    indentLevel, dir.Name, dir.LastAccessTime);
    // get all the directories in the current directory
    // and call this method recursively on each
    DirectoryInfo[] directories = dir.GetDirectories( );
    foreach (DirectoryInfo newDir in directories)
    {
      dirCounter++; // increment the counter
      ExploreDirectory(newDir);
    }
    indentLevel--; // pop a directory level
  }
  // static member variables to keep track of totals
  // and indentation level
  static int dirCounter = 1;
  static int indentLevel = -1; // so first push = 0
}
```

14.1.2 Lucrul cu fişierele: clasele *FileInfo* şi *File*

Un obiect *DirectoryInfo* poate de asemenea să returneze o colecţie a tuturor fişierelor conţinute, sub forma unor obiecte de tip *FileInfo*. Înrudită cu clasa *FileInfo* (care are membri nestatici) este clasa *File* (care are doar membri statici). Tabelele 14.3 şi 14.4 conţin metodele pentru fiecare clasă:

Tabelul 14.3: Metode ale clasei *File*.

Metoda	Explicaţie
AppendText()	Creează un obiect *StreamWriter* care adaugă text la fişierul specificat
Copy()	Copiază un fişier existent într–un alt fişier
Create()	Creează un fişier în calea specificată
CreateText()	Creează un *StreamWriter* care scrie un nou fişier text
Delete()	Şterge fişierul specificat
Exists()	Returnează true dacă fişierul corespunzător există
GetAttributes() SetAttributes()	Returnează / setează *FileAttributes* pentru fişierul specificat
GetCreationTime() SetCreationTime()	Returnează / setează data şi timpul creării pentru fişierul specificat
GetLastAccessTime() SetLastAccessFile()	Returnează sau setează timpul ultimului acces la fişier
GetLastWriteTime() SetLastAccessTime()	Returnează / setează timpul ultimei modificări a fişierului
Move()	Mută un fişier la o nouă locaţie; poate fi folosit pentru redenumire
OpenRead()	Metodă returnând un *FileStream* pe un fişier
OpenWrite()	Crează un *Stream* de citire / scriere

Tabelul 14.4: Metode ale clasei *FileInfo*.

Metoda	Explicaţie
Attibutes	Returnează sau setează atributele fişierului curent
CreationTime	Returnează sau setează timpul de creare al fişierului curent
Directory	Returnează o instanţă a directorului curent
Exists	true dacă fişierul există
Extension	Returnează extensia fişierului
FullName	Calea absolută până la fişierul curent
LastAccessTime	Returnează sau setează timpul ultimului acces
LastWriteTime	Returnează sau setează timpul când s–a modificat ultima oară fişierul curent
Length	Returnează dimensiunea fişierului
Name	Returnează numele instanţei curente
AppendText	Crează un *StreamWriter* care va permite adăugarea

Tabelul 14.4 (continuare)

Metoda sau proprietatea	Explicaţie
	la fişier
CopyTo()	Copiează fişierul curent într–un alt fişier
Create()	Crează un nou fişier
Delete()	Şterge un fişier
MoveTo()	Mută un fişier la o locaţie specificată; poate fi folosită pentru redenumire
OpenRead() OpenText() OpenWrite()	Crează un fişier read–only respectiv StreamReader(text) sau *FileStream*(read–write)

Exemplul anterior este modificat pentru a afişa informaţii despre fişiere: numele, dimensiunea, data ultimei modificări:

```
using System;
using System.IO;
class Tester
{
  public static void Main( )
  {
    Tester t = new Tester( );
    // choose the initial subdirectory
    string theDirectory = @''c:\WinNT'';
    // call the method to explore the directory,
    // displaying its access date and all
    // subdirectories
    DirectoryInfo dir = new DirectoryInfo(theDirectory);
    t.ExploreDirectory(dir);
    // completed. print the statistics
    Console.WriteLine(``\n\n{0} files in {1} directories found.\n'',
    fileCounter,dirCounter);
  }
  // Set it running with a directoryInfo object
  // for each directory it finds, it will call
  // itself recursively
  private void ExploreDirectory(DirectoryInfo dir)
  {
    indentLevel++; // push a directory level
    // create indentation for subdirectories
    for (int i = 0; i < indentLevel; i++)
```

```
    Console.Write(‘‘ ‘‘); // two spaces per level
    // print the directory and the time last accessed
    Console.WriteLine(‘‘[{0}] {1} [{2}]\n’’, indentLevel, dir.Name,
      dir.LastAccessTime);
    // get all the files in the directory and
    // print their name, last access time, and size
    FileInfo[] filesInDir = dir.GetFiles( );
    foreach (FileInfo file in filesInDir)
    {
      // indent once extra to put files
      // under their directory
      for (int i = 0; i < indentLevel+1; i++)
      Console.Write(‘‘ ’’); // two spaces per level
      Console.WriteLine(‘‘{0} [{1}] Size: {2} bytes’’, file.Name, file.LastWriteTime
      file.Length);
      fileCounter++;
    }
    // get all the directories in the current directory
    // and call this method recursively on each
    DirectoryInfo[] directories = dir.GetDirectories( );
    foreach (DirectoryInfo newDir in directories)
    {
      dirCounter++; // increment the counter
      ExploreDirectory(newDir);
    }
    indentLevel--; // pop a directory level
  }
  // static member variables to keep track of totals
  // and indentation level
  static int dirCounter = 1;
  static int indentLevel = -1; // so first push = 0
  static int fileCounter = 0;
}
```

Exemplul următor nu introduce clase noi, ci doar exemplifică crearea unui director, copierea de fişiere în el, ştergerea unora dinre ele şi în final ştergerea directorului:

```
using System;
using System.IO;
class Tester
{
```

```
  public static void Main( )
  {
    // make an instance and run it
    Tester t = new Tester( );
    string theDirectory = @''c:\test\media'';
    DirectoryInfo dir = new DirectoryInfo(theDirectory);
    t.ExploreDirectory(dir);
  }
  // Set it running with a directory name
  private void ExploreDirectory(DirectoryInfo dir)
  {
    // make a new subdirectory
    string newDirectory = ``newTest'';
    DirectoryInfo newSubDir =
    dir.CreateSubdirectory(newDirectory);
    / get all the files in the directory and
    // copy them to the new directory
    FileInfo[] filesInDir = dir.GetFiles( );
    foreach (FileInfo file in filesInDir)
    {
      string fullName = newSubDir.FullName + ``\\'' + file.Name;
      file.CopyTo(fullName);
      Console.WriteLine(``{0} copied to newTest'', file.FullName);
    }
    // get a collection of the files copied in
    filesInDir = newSubDir.GetFiles( );
    // delete some and rename others
    int counter = 0;
    foreach (FileInfo file in filesInDir)
    {
      string fullName = file.FullName;
      if (counter++ %2 == 0)
      {
file.MoveTo(fullName + ``.bak'');
Console.WriteLine(``{0} renamed to {1}'',
fullName,file.FullName);
      }
      else
      {
file.Delete( );
Console.WriteLine(``{0} deleted.'',
```

```
fullName);
      }
    }
    newSubDir.Delete(true); // delete the subdirectory
  }
}
```

14.2 Citirea şi scrierea datelor

Clasele disponibile pentru lucrul cu stream–uri sunt:

Tabelul 14.5: Clase pentru lucrul cu stream–uri

Clasa	Descriere
Stream	Manipulare generică a unei secvenţe de octeţi; clasă abstractă
BinaryReader	Citeşte tipuri de date primitive ca voalori binare într–o codificare specifică
BinaryWriter	Scrie tipuri de date primitive într–un flux binar; de asemenea scrie string–uri folosind o anumită codificare
BufferedStream	Ataşează un buffer unui stream de intrare / ieşire. Clasă *sealed*
FileStream	Ataşează un stream unui fişier, permiţând operaţii sincrone sau asincrone de citire şi scriere.
MemoryStream	Crează un stream pentru care citirea / stocarea de date se face în memorie
NetworkStream	Creează un stream folosind TCP/IP
TextReader	Permite citire de caractere, în mod secvenţial. Clasă abstractă.
TextWriter	Permite scriere secvenţială într–un fişier. Clasă abstractă.
StreamReader	Implementează o clasă *TextReader* care citeşte caractere dintr–un stream folosind o codificare particulară
StreamWriter	Implementează o clasă *TextWriter* care scrie caractere într–un stream folosind o codificare particulară
StringReader	Implementează un TextReader care citeşte dintr–un string
StringWriter	Scrie informaţie într–un string. Informaţia este stocată într–un StringBuilder

14.2.1 Clasa *Stream*

Clasa *Stream* este o clasă abstractă din care se derivează toate celelalte clase de lucru cu stream–uri. Metodele conţinute permit citire de octeţi,

închidere, golire de buffer, etc. Pe lângă acestea, clasa *Stream* permite atât operaţii sincrone, cât şi asincrone. Într–o operaţie de intrare / ieşire sincronă, dacă se începe o operaţie de citire sau scriere dintr–un / într–un flux, atunci programul va trebui să aştepte până când această operaţie se termină. Sub platforma .NET se poate face însă op operaţie de intrare / ieşire în mod asincron, astfel permiţând altor fire de execuţie să se execute.

Metodele folosite pentru a în cepe o astfel de intrare asincronă sunt: *BeginRead(), BeginWrite(), EndRead(), EndWrite()*. O dată ce o astfel de operaţie se termină, o metodă callback se va executa automat.

O listă a metodelor şi proprietăţilor care sunt definite în clasa *Stream* este dată în tabelul 14.6.

Tabelul 14.6: Metode şi proprietăţi ale clasei Stream

Clasa	Descriere
BeginRead()	Începe o citire asincronă
BeginWrite()	Începe o scriere asincronă
Close()	Închide stream–ul curent şi eliberează resursele asociate cu el (socket–uri, file handles, etc)
EndRead()	Aşteaptă pentru o terminare de citire asincronă.
EndWrite()	Aşteaptă pentru o terminare de scriere asincronă.
Flush()	Când este suprascrisă în tr–o clasă derivată, goleşte bufferele asociate straem–ului curent şi determină scrierea lor.
Read()	Când este suprascrisă într–o clasă derivată, citeşte o secvenţă de octeţi şi incrementează indicatorul de poziţie curentă în stream
ReadByte()	Citeşte un byte din stream şi incrementează indicatorul de poziţie curent; dacă este la sfârşit de fişier, returnează -1
Seek()	Când este suprascrisă într–o clasă derivată, setează poziţia curentă în interiorul stream–ului
SetLength()	Când este suprascrisă într–o clasă derivată, setează lungimea stream–ului curent
Write()	Când este suprascrisă într–o clasă derivată, scrie o secvenţă de octeţi în stream–ul curent şi incrementează corespunzător indicatorul poziţiei curente în stream
WriteByte()	Scrie un byte la poziţia curentă din stream şi incrementează indicatorul de poziţie curentă
CanRead()	Când este suprascrisă într–o clasă derivată, returnează o valoarea care indică dacă stream–ul curent poate fi citit
CanWrite()	Când este suprascrisă într–o clasă derivată, returnează

Tabelul 14.6 (continuare)

Metoda	Descriere
	o valoarea care indică dacă stream–ul curent suportă scriere
CanSeek	Când este suprascrisă într–o clasă derivată, returnează o valoarea care indică dacă se poate face poziţionare aleatoare în stream–ul curent
Length	Când este suprascrisă într–o clasă derivată, returnează dimensiunea în octeţi a fişierului
Position	Când este suprascrisă într–o clasă derivată, returnează sau setează poziţia curentă în interiorul stream–ului

14.2.2 Clasa *FileStream*

Există mai multe metode de obţinere a unui obiect de tip *FileStream*. Clasa prezintă nouă constructori supraîncărcaţi. Enumerarea *FileMode* este folosită pentru a se specifica modul de deschidere a unui stream: (*Append, Create, CreateNew, Open, OpenOrCreate, Truncate*).

Exemplu: mai jos se creează un fişier nou (dacă nu există) sau se suprascrie unul existent:

```
FileStream f = new FileStream( @''C:\temp\a.dat'', FileMode.Create );
```

De asemenea se mai poate obţine o instanţă a clasei *FileStream* din clasa *File*:

```
FileStream g = File.OpenWrite(@''c:\temp\test.dat'');
//deschidere doar pentru citire
```

Asemănător, se poate folosi o instanţă a clasei *FileInfo*:

```
FileInfo fi = new FileInfo(@''c:\temp\test.dat'');
FileStream fs = fi.OpenRead();
//deschidere doar pentru citire
```

14.2.3 Clasa *MemoryStream*

Un *MemoryStream* îşi ia datele din memorie, văzută ca un vector de octeţi. Există şapte constructori pentru această clasă, dar care pot fi grupaţi în două categorii.

Primul tip de constructor preia un array de octeţi pentru care poate face citire şi opţional scriere. Caracteristic este că tabloul nu va putea fi redimensionat:

```
byte[] b = {1, 2, 3, 4};
MemoryStream mem = new MemoryStream(b);
```

O altă variantă de constructor nu primeşte un vector de octeţi, dar va putea scrie într–un tablou redimensionabil. Opţional, se specifică un *int* ca parametru al constructorului care determină dimensiunea iniţială a tabloului de octeţi. Datele sunt adăugate folosind metoda *Write()*:

```
using System;
using System.IO;

public class MemTest
{
  public static void Main()
  {
    MemoryStream mem = new MemoryStream();
    byte[] bs = {1, 2, 3, 4, 5, 6};
    mem.Write(bs, 0, bs.Length);
    mem.Seek(3, SeekOrigin.Begin);
    byte b = (byte)mem.ReadByte();
    Console.WriteLine(‘‘Value: {0}’’, b.ToString());
    //se va afisa 4
  }
}
```

E de preferat să se utilizeze obiecte de tip *MemoryStream* în scopul de a accesa informaţia din memoria RAM, mai degrabă decât de pe disc sau din reţea. De exemplu se poate încărca un fişier de pe disc în memorie, astfel încât analiza lui se poate face mai repede.

14.2.4 Clasa *BufferedStream*

C# pune la dispoziţie o clasă *BufferedStream* pentru a asigura o zonă tampon în cazul operaţiilor de intrare–ieşire. Constructorul acestei clase primeşte o instanţă a clasei *Stream*.

Exemplu:

```
FileStream fs = new FileStream(@’’c:\temp\a.dat’’, FileMode.Open);
BufferedStream bs = new BufferedStream(fs);
```

De menţionat aici metoda *Flush()*, care forţează golirea bufferului asociat stream–ului.

14.2.5 Clasele *BinaryReader* şi *BinaryWriter*

Cele două clase sunt folosite pentru a accesa date mai complexe decât un byte: de exemplu, pentru manipularea datelor de tip boolean, sau Decimal, sau int cu semn pe 64 de biţi.

Tabelul 14.7 conţine metodele puse la dispoziţie de către clasa *BinaryWriter*:

Tabelul 14.7: Metodele clasei *BinaryReader*

Metoda	Descriere
PeekChar()	Returnează următorul caracter disponibil, fără a avansa indicatorul de poziţie curentă
Read()	Citeşte caractere din flux şi avansează poziţia curentă din acel stream
ReadBoolean()	Citeşte un Boolean din stream şi avansează poziţia curentă cu un byte
ReadBytes()	Citeşte un număr precizat de octeţi într–un vector şi avansează poziţia curentă
ReadChar()	Citeşte următorul caracter şi avansează poziţia curentă corespunzător cu codificarea folosită pentru caracter
ReadChars()	Citeşte mai multe caractere într–un vector şi avansează poziţia curentă cu numărul de caractere dimensiunea de reprezentare pentru caracter
readDecimal()	Citeşte un decimal şi avansează poziţia curentă cu 16 octeţi
ReadDouble()	Citeşte o variabilăîn virgulă mobilă şi avansează cu 8 octeţi
ReadInt16()	Citeşte un întreg cu semn pe 16 biţi şi avansează cu 2 octeţi
ReadInt32()	Citeşte un întreg cu semn pe 32 de biţi şi avansează cu 4 octeţi
ReadInt64()	Citeşte un întreg cu semn pe 64 de biţi şi avansează cu 8 octeţi
ReadSByte()	Citeşte un byte cu semn şi avansează cu un byte
ReadSingle()	Citeşte o valoare în virgulă mobilă pe 4 octeţi şi avansează poziţia curentă cu 4 octeţi
ReadString()	Citeşte un string, prefixat cu lungimea sa, codificată ca un întreg reprezentat pe grupe de c&ate 7 biţi (MSDN)
ReadUInt16	Citeşte un întreg fără semn reprezentat pe 16 biţi şi avansează cu 2 octeţi
ReadUInt32	Citeşte un întreg fără semn reprezentat pe 32 de biţi şi avansează cu 4 octeţi
ReadUInt64	Citeşte un întreg fără semn reprezentat pe 64 de biţi şi avansează cu 8 octeţi

Clasa *BinaryWriter* are o metodă *Write()* supraîncărcată, care poate fi apelată pentru scrierea diferitelor tipuri de date. O menţiune la scrierea de string–uri şi de caractere / ţectori de caractere: caracterele pot fi codificate în mai multe moduri (ex: ASCII, UNICODE, UTF7, UTF8), codificare care se poate transmite ca argument pentru constructor. A se vedea MSDN şi RFC–urile de pe Interner.

14.2.6 Clasele *TextReader, TextWriter* şi descendentele lor

Pentru manipularea şirurilor de caractere aflate în fişiere, dar nu numai, C# pune la dispoziţie clasele abstracte *TextReader, TextWriter*. Clasa *TextReader* are subclasele neabstracte *StreamReader* şi *StringReader*. Clasa *TextWriter* are subclasele neabstracte *StreamWriter, StringWriter, System.Web.HttpWriter, System.Web.UI.HtmlTextWriter, System.CodeDom.Compiler.IndentedTextWriter*.

Descrieri şi exemple sunt date mai jos:

1. Clasele *StreamReader* şi *StreamWriter* - sunt folosite pentru a cit sau scrie şiruri de caractere. Un obiect de tip *StreamReader* se poate obţine via un constructor:

   ```
   StreamReader sr = new StreamReader(@''C:\temp\siruri.txt'');
   ```

   sau pe baza unui obeiect de tip *FileInfo*:

   ```
   FileInfo fi = new FileInfo(@''c:\temp\siruri.txt'');
   StreamReader sr = fi.OpenText();
   ```

   Obiectele de tip *StreamReader* pot citi câte o linie la un moment dat folosind metoda *ReadLine()*.

   Exemplu:

   ```
   using System;
   using System.IO;
   class Test
   {
     static void Main()
     {
       StreamReader sr = new StreamReader(``c:\temp\siruri.txt'');
       String line;
       do
   ```

```
    {
      line = sr.ReadLine();
      Console.WriteLine(line);
      //daca line==null, atunci se va afisa o linie vida
    }while( line!=null);
  }
}
```

2. Clasele *StringReader* şi *StringWriter* - permit ataşarea unor fluxuri la şiruri de caractere, folosite pe post de surse de date.

   Exemplu:

```
string myString = ‘‘1234567890’’;
StringReader sr = new StringReader(myString);
```

```
using System;
using System.IO;
using System.Xml;
class Test
{
  static void Main()
  {
    XmlDocument doc = new XmlDocument();
    String entry = ‘‘<book genre=’biography’’’ +
    ‘‘ ISBN=’12345678’><title>Yeager</title>’’ +
    ‘‘</book>’’;
    doc.LoadXml(entry);//salvare in doc. xml

    StringWriter writer = new StringWriter();
    doc.Save(writer);
    string strXml = writer.ToString();
    Console.WriteLine(strXml);
  }
}
```

   va afişa:

```
<?xml version=’’1.0’’ encoding=’’utf-16’’>
<book genre=’’biography’’ ISBN=’’12345678’’>
  <title>Yeager</title>
</book>
```

În loc ca salvarea din documentul xml să se facă într–un fişier text, se face într–un obiect de tip StringWriter(), al cărui conţinut se va afişa.

3. *IndentedTextWriter* defineşte metode care inserează tab–uri şi păstrează evidenţa niveluilui de identare. Este folosit de deriuvări ale clasei *CodeDom*, folosită pentru generare de cod.

4. *HtmlTextWriter* scrie o secvenţă HTML pe o pagină Web. Este folosit de exemplu în script-uri C#, în cazul aplicaţiilor ASP.NET

5. *HttpWriter* - prea puţină informaţie!!!

14.3 Operare sincronă şi asincronă

Exemplele folosite până acum au folosit un mod de operare sincron, adică atunci când se face o operaţie de intrare/ieşire, întregul program este blocat până când se tranzitează toate datele specificate. Stream–urile C# permit şi acces asincron, permiţând altor fire de execuţie să fie rulate. Pentru semnalarea începutului unei operaţii de citire sau scriere asincrone, se folosesc *BeginRead()* şi *BeginWrite()*.

Metoda *BeginRead* are prototipul:

```
public override IAsyncResult BeginRead(
  byte[] array, int offset, int numBytes,
  AsyncCallback userCallback, object stateObject
);
```

Vectorul de bytes reprezintă bufferul în care se vor citi datele; al doilea şi al treilea parametru determină byte–ul la care să se va scrie, respectiv numărul maxim de octeţi care se vor citi. Al patrulea parametru este un delegat, folosit pentru a semnala (mai exact, a face nişte prelucrări) sfârşitul citirii. Se poate transmite *null* pentru acest parametru, dar programul nu va fi notificat de terminarea citirii. Ultimul parametru este un obiect care va fi folosit pentru a distinge între această cerere de citire asincronă şi altele.

```
using System;
using System.IO;
using System.Threading;
using System.Text;
public class AsynchIOTester
{
  private Stream inputStream;
```

```
// delegated method
private AsyncCallback myCallBack;
// buffer to hold the read data
private byte[] buffer;
// the size of the buffer
const int BufferSize = 256;
// constructor
AsynchIOTester( )
{
  // open the input stream
  inputStream =
  File.OpenRead(
  @"C:\test\source\AskTim.txt");
  // allocate a buffer
  buffer = new byte[BufferSize];
  // assign the call back
  myCallBack =
  new AsyncCallback(this.OnCompletedRead);
}
public static void Main( )
{
  // create an instance of AsynchIOTester
  // which invokes the constructor
  AsynchIOTester theApp =
  new AsynchIOTester( );
  // call the instance method
  theApp.Run( );
}
void Run( )
{
  inputStream.BeginRead(
  buffer, // holds the results
  0, // offset
  buffer.Length, // (BufferSize)
  myCallBack, // call back delegate
  null); // local state object
  // do some work while data is read
  for (long i = 0; i < 500000; i++)
  {
    if (i%1000 == 0)
    {
```

```
Console.WriteLine("i: {0}", i);
      }
    }
  }
  // call back method
  void OnCompletedRead(IAsyncResult asyncResult)
  {
    int bytesRead =
    inputStream.EndRead(asyncResult);
    // if we got bytes, make them a string
    // and display them, then start up again.
    // Otherwise, we're done.
    if (bytesRead > 0)
    {
      String s =
      Encoding.ASCII.GetString(buffer, 0, bytesRead);
      Console.WriteLine(s);
      inputStream.BeginRead(
      buffer, 0, buffer.Length, myCallBack, null);
    }
  }
}
```

Ieşirea ar fi:

```
i: 47000
i: 48000
i: 49000
Date: January 2001
From: Dave Heisler
To: Ask Tim
Subject: Questions About O'Reilly
Dear Tim,
I've been a programmer for about ten years. I had heard of
O'Reilly books,then...
Dave,
You might be amazed at how many requests for help with
school projects I get;
i: 50000
i: 51000
i: 52000
```

Cele două fire de execučtie lucrează deci concurent.

14.4 Stream–uri Web

C# conţine clase gândite pentru a uşura interoperarea cu web–ul. Aducerea informaţiei de pe web se face în doi paşi: primul pas constă în a face o cerere de conectare la un server; dacă cererea se poate face, atunci în pasul al doilea constă în obţinerea informaţiei propriu–zise de la server. Cei doi paşi se fac respectiv cu clasele *HttpWebRequest*, respectiv *HttpWebResponse*

Un obiect *HttpWebRequest* poate fi obţinut prin metoda statică *Create()* din clasa *WebRequest*:

```
string page = ‘‘http://www.cetus-links.org/index.html’’;
HttpWebRequest webRequest = (HttpWebRequest)WebRequest.Create(page);
```

Pe baza obiectului *HttpWebRequest* obţinut se va crea un obiect *HttpWebResponse*:

```
HttpWebResponse webResponse = (HttpWebResponse)webRequest.GetResponse();
```

În final, se obţine un stream prin metoda *GetResponseStream()*:

```
StreamReader streamReader =
  new StreamReader(webResponse.GetResponseStream(), Encoding.ASCII);
```

14.5 Serializarea

Serializarea reprezintă posibilitatea de a trimite un obiect printr–un stream. Pentru aceasta, C# foloseşte metodele *Serialize()* şi *Deserialize()* ale clasei *BinaryFormatter*.

Metoda *Serialize()* are nevoie de 2 parametri: un stream în care să scrie şi obiectul pe care să îl serializeze. Metoda de deserializare cere doar un stream din care să citească, şi din care să refacă un object (care poate fi convertit la tipul corespunzător).

14.5.1 Crearea unui obiect serializabil

Pentru ca o clasă definită de utilizator să suport serializarea, este nevoie de a preciza atributul *[Serializable]* în faţa declaraţiei clasei respective, atribut definit în clasa *System.SerializableAttribute*. Tipurile primitive sunt automat serializabile, iar dacă tipul definit de utilizator conţine alte tipuri, atunci acestea la rândul lor trebuie să poată fi serializate.

Exemplu:

```
using System;
[Serializable]
public class BookRecord
{
  public String title;
  public int asin;
  public BookRecord(String title, int asin)
  {
    this.title = title;
    this.asin = asin;
  }
}
```

14.5.2 Serializarea

Codul pentru serializarea unui obiect de tipul declarat mai sus este:

```
using System;
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;
public class SerializeObject
{
  public static void Main()
  {
    BookRecord book = new BookRecord(
    "Building Robots with Lego Mindstorms",
    1928994679);
    FileStream stream = new FileStream(@"book.obj",
    FileMode.Create);
    BinaryFormatter bf = new BinaryFormatter();
    bf.Serialize(stream, book);
    stream.Close();
  }
}
```

14.5.3 Deserializarea unui obiect

Deserializarea se face astfel:

```
using System;
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;
```

```
public class DeserializeObject
{
  public static void Main()
  {
    FileStream streamIn = new FileStream(
    @"book.obj", FileMode.Open);
    BinaryFormatter bf = new BinaryFormatter();
    BookRecord book =
    (BookRecord)bf.Deserialize(streamIn);
    streamIn.Close();
    Console.Write(book.title + " " + book.asin);
  }
}
```

14.5.4 Date tranziente

Uneori este nevoie ca anumite câmpuri ale unui obiect să nu fie salvate: parola unui utilizator, numărul de cont al unui client, etc. Acest lucru se face specificând atributul *[NonSerialized]* pentru câmpul respectiv:

```
[NonSerialized] int secretKey;
```

14.5.5 Operaţii la deserializare

Uneori este nevoie ca o deserializare să fie automat urmată de o anumită operaţie. De exemplu, un câmp care nu a fost salvat (tranzient) va trebui să fie refăcut în mod calculat. Pentru acest lucru, C# pune la dispoziţie interfaţa *IDeserializationCallback*, pentru care trebuie să se implementeze metoda *OnDeserialization*. Această metodă va fi apelată automat de către CLR atunci când se va face deserializarea obiectului.

Exemplul de mai jos ilustrează acest mecanism:

```
using System;
using System.Runtime.Serialization;
[Serializable]
public class BookRecord: IDeserializationCallback
{
  public String title;
  public int asin;
  [NonSerialized] public int sales_rank;
  public BookRecord(String title, int asin)
  {
```

```
    this.title = title;
    this.asin = asin;
    sales_rank = GetSalesRank();
  }
  public int GetSalesRank()
  {
    Random r = new Random();
    return r.Next(5000);
  }
  public void OnDeserialization(Object o)
  {
    sales_rank = GetSalesRank();
  }
}
```

Mecanismul poate fi folosit în special atunci când serializarea unui anumit câmp, care ar duce la mult spaţiu de stocare consumat, ar putea fi calculat în mod automat la deserializare (spaţiu vs. procesor).

Bibliografie

[1] *C# in depth*, Manning, Jon Skeet, 2008

[2] *Programming C#*, O'Reilly, Jesse Liberty, 4th edition, 2005

[3] *Microsoft C# 2005 Step by Step*, John Sharp, Microsoft Press, 2005

[4] *Core C# and .NET*, Stephen C. Perry, Prentice Hall, 2005

[5] *Pro ASP.NET in C# 2005*, Mathew MacDonald and Mario Szpuszta, Apress, 2005

[6] *C# Language Specification*, ECMA TC39/TG2, Octombrie 2002

[7] *Professional ADO.NET Programming*, Wrox, 2001

[8] *LINQ for Visual C# 2008*, Fabio Claudio Ferracchiati, Apress, 2008

[9] *PLINQ*, Daniel Moth, 2009.